Universidad Zaragoza
1542

Proyecto Fin de Carrera

SIMULADOR DE ESCENARIOS ROBÓTICOS CON CAPACIDAD MULTIROBOT : MEMORIA

Autor

José Javier Pérez Rubio

Director

Antonio Romeo Tello

Escuela de Ingeniería y Arquitectura

2012

ESCUELA DE INGENIERÍA Y ARQUITECTURA

UNIVERSIDAD DE ZARAGOZA

HOJA DE IDENTIFICACIÓN

PFC 3º ELECTRÓNICA 11/12

SIMULADOR DE ESCENARIOS ROBÓTICOS CON CAPACIDAD MULTIROBOT

MEMORIA

DIRECTOR: Antonio Romeo Tello
DEPARTAMENTO DE INGENIERÍA DE SISTEMAS Y AUTOMÁTICA
CORREO ELECTRÓNICO: romeo@unizar.es

AUTOR:
José Javier Pérez Rubio
TITULACIÓN: Ingeniero Técnico Industrial
ESPECIALIDAD: Electrónica Industrial
CORREO ELECTRÓNICO: 554759@unizar.es

FECHA Y FIRMA:

José Javier Pérez Rubio

Zaragoza, 27 de agosto de 2012

RESUMEN DEL PFC

El **objetivo** que se pretende con este Proyecto es la elaboración de un completo constructor de entornos virtuales para la explotación y simulación de escenarios robóticos industriales con capacidad multirobot. Se partirá de la versión mono-robot de RobotScene con el fin de transformarla en una versión multi-robot.

Los **requisitos** de diseño de la nueva aplicación serán:

- Una ventana de programación por cada uno de los tres robots. Estas ventanas deberán ejecutarse independientemente, afectando cada una sólo a su correspondiente robot.
- La aplicación deberá permitir la ejecución de tareas en las que intervengan varios robots en el mismo escenario de forma simultánea.
- Los robots serán capaces de comunicarse mediante la emulación de un cableado de E/S.

La **implementación** de los requisitos anteriores se consigue mediante

→ Creación y gestión de ventanas de programación mediante la producción de una instancia de ventana por cada robot

→ Consecución de la concurrencia de robots mediante la creación de varios hilos de ejecución.

→ Comunicación entre robots a través de una ventana de cableado

En la práctica, llevar a cabo y conseguir todo lo anterior ha supuesto:

- ✓ La modificación de la interfaz de usuario de la versión mono-robot
- ✓ La modificación del escenario robótico, de manera que sea capaz de soportar la ejecución de varios scripts robóticos de manera simultánea.
- ✓ La creación de una ventana de cableado interactiva que permita la gestión de la comunicación entre robots

ÍNDICE DE LA MEMORIA

1 OBJETIVOS DEL PROYECTO Y JUSTIFICACIÓN 5
2 ALCANCE 6
3 ANTECEDENTES 7
3.1 ANTECEDENTES DE LA ROBÓTICA 8
3.2 DESARROLLO DE LA ROBÓTICA 12
3.3 DEFINICIÓN DE ROBOT INDUSTRIAL 15
3.4 CLASIFICACIÓN DE LOS ROBOTS INDUSTRIALES 17
3.5 SIMULADORES ROBÓTICOS 18
3.5.1 Simuladores de robots móviles 19
3.5.2 Simuladores de robots industriales 22
3.6 ROBOTSCENE COMO PUNTO DE PARTIDA 27
3.6.1 Características de RobotScene (versión mono-robot) 27
3.6.2 Análisis de la solución seleccionada en RobotScene (versión mono-robot) 28
4 REQUISITOS DE DISEÑO 38
5 ANÁLISIS DE SOLUCIONES 41
5.1 CREACIÓN Y GESTIÓN DE LAS VENTANAS DE PROGRAMACIÓN. 42
5.2 CONSECUCIÓN DE LA CONCURRENCIA 44
5.3 COMUNICACIÓN ENTRE ROBOTS 46
6 DESCRIPCIÓN DE LA APLICACIÓN 48
6.1 DESCRIPCIÓN DE LA INTERFAZ DE USUARIO 49
6.2 CREACIÓN DEL ESCENARIO 55
6.3 EXPLOTACIÓN DEL ESCENARIO 59
6.4 ESTRUCTURA DEL CÓDIGO FUENTE 73
6.5 DESCRIPCIÓN DEL LENGUAJE DE PROGRAMACIÓN DE ROBOTS 100
7 RESULTADOS FINALES 107
7.1 SECUENCIA CRONOLÓGICA DE ACTUACIONES 108
7.2 PROBLEMAS Y SOLUCIONES 112
7.3 COMENTARIOS FINALES 115
8 REFERENCIAS BIBLIOGRÁFICAS E INFORMÁTICAS 116
8.1 BIBLIOGRAFÍA / LINKOGRAFÍA 116
8.2 PROGRAMAS DE CÁLCULO Y SOFTWARE UTILIZADOS 117
9 GLOSARIO DE TÉRMINOS 118
9.1 DEFINICIONES 118
9.2 ABREVIATURAS 122

1 OBJETIVOS DEL PROYECTO Y JUSTIFICACIÓN

El objetivo que se pretende con este proyecto es la elaboración de un completo constructor de entornos virtuales para la explotación y simulación de escenarios robóticos industriales con capacidad multirobot.

El usuario final del entorno virtual podrá crear escenarios robóticos, en los cuales es posible la programación y simulación de tareas robóticas industriales de manera sencilla, visual e interactiva. El programa permite la creación de escenarios que contengan varios sólidos que formarán parte del entorno, y los cuales podrán ser manipulados posteriormente por uno o varios robots industriales.

Para la posterior explotación y simulación de las tareas robóticas se ha habilitado de un sistema de programación de robots. Dicho sistema de programación permite la creación de tareas robóticas que podrán ser ejecutadas por uno o varios robots de manera independiente o de forma simultánea. Para ello se ha habilitado un lenguaje de programación de corte robótico, sencillo, estructurado y cuya sintaxis está basada en Pascal. Se permite de este modo la realización de simulaciones en las cuales intervengan de forma directa y simultánea varios robots que tendrán plena capacidad de comunicación entre ellos. Para lograr esta comunicación el programa cuenta con un editor de cableado que pretende simular un cableado real mediante los módulos de entradas y salidas de los robots industriales.

Si bien es cierto que actualmente existen infinidad de simuladores robóticos, también hay que indicar que todos ellos adolecen de poca o nula flexibilidad en cuanto al número de robots o en cuanto al tipo de esos robots. Por ejemplo, existen multitud de simuladores robóticos creados únicamente para un tipo de robot debido en gran parte a que estos simuladores son creados por la propia empresa fabricante del robot. Otros simuladores son sistemas más abiertos y permiten el uso de varios tipos de robots; sin embargo, presentan el inconveniente de que el usuario debe recompilar la aplicación, lo que repercute en la facilidad de manejo del simulador al ser necesaria la utilización de componentes externos (compiladores).

Con otra filosofía diferente fue creado **RobotScene** que permite la construcción visual e interactiva tanto del robot como de los elementos que toman parte en el escenario virtual en el que transcurre la acción. Además, se trata de una aplicación auto contenida, no precisa de la instalación de compiladores externos ni de ningún otro software adicional, y en la cual pueden crearse tantos tipos de robots como el usuario pueda imaginar. A pesar de todo esto, **RobotScene** contiene una gran limitación debido a que imposibilita la simulación de tareas con más de un robot.

Partiendo de esta base y con la intención de llegar un poco más lejos, el objetivo del proyecto es la mejora de **RobotScene** de manera que se elimine su limitación y se pueda implementar la simulación de escenarios robóticos con capacidad multirobot.

2 ALCANCE

La aplicación que se ha desarrollado está enfocada claramente hacia la simulación de escenarios robóticos, lo que hace imprescindible para su utilización ciertos conocimientos tanto de robótica industrial como de programación. Aunque no es necesario que el usuario posea profundos conocimientos de ambas materias, si es cierto que hay ciertos conceptos básicos que el usuario debe tener claros: debe ser capaz de realizar asignaciones de parámetros de Denavit Hartenberg, conocer las acciones básicas que puede realizar un robot, saber qué son las matrices de transformación homogénea y cómo se combinan, saber controlar el flujo de ejecución de un programa: estructuras condicionales, bucles, subrutinas, etc…

La aplicación persigue conformar un simulador flexible, en el cual el usuario no se vea obligado a la utilización de un tipo determinado de robot, ni se le imponga el uso de lenguajes de programación robóticos comerciales. Así mismo, el programa facilitará al usuario profundizar en materias robóticas y acercarse a la realidad industrial sin la necesidad de disponer de un robot. Permite, de esta forma, implementar la definición del robot en papel (obviamente, esta parte es ineludible) en un entorno virtual donde se puede visualizar tanto el robot, como sus movimientos y programar aplicaciones propias.

Por todo lo anterior consideramos que el proyecto tiene un alcance claramente educativo y que facilitará a estudiantes la comprensión y familiarización de las materias robóticas.

3 ANTECEDENTES

Aunque el robot es considerado como una máquina compleja, se trata de un aparato ampliamente conocido por la sociedad. Esto es debido, en gran parte, a que la literatura y el cine de ciencia ficción han dado a conocer a través de sus obras el concepto de robot. A pesar de que la inmensa mayoría de las veces el robot haya sido mitificado, todos podemos hacernos una idea de lo que es un robot industrial. Lo hemos visto algunas veces en fotografías o videos que circulan por revistas, televisión o por internet hablando sobre noticias o artículos de industria, con lo que podemos imaginar qué es y para qué funciona.

En este apartado conoceremos un poco de historia de la robótica, desde los primeros autómatas hasta los robots actuales. También estudiaremos someramente la clasificación de los robots industriales. Por último analizaremos las características de algunos de los simuladores robóticos más conocidos.

3.1 ANTECEDENTES DE LA ROBÓTICA

A lo largo de la historia, el ser humano ha intentado crear dispositivos que fueran capaces de imitar movimientos del cuerpo humano. Los griegos llamaban a estas máquinas autómatas. Una de las definiciones actuales de autómata dada por la RAE es: *máquina que imita la figura y los movimientos de un ser animado*.

Los primeros mecanismos automáticos aparecen en la antigua Etiopía, en el año 1500 a. C., Amenhotep, hermano de Hapu, construyó una estatua de Memon, el rey de Etiopía, que emitía sonidos cuando la iluminaban los rayos del sol al amanecer.

En torno al año 500 a.C. se construyeron en China una urraca voladora de madera y un caballo de madera que saltaba. Herón de Alejandría (85 d.C.) describió múltiples mecanismos inventados con fines lúdicos. Posteriormente, los árabes (siglos VII) crearon mecanismos con utilidades prácticas tales como sistemas automáticos para beber o lavarse.

También en la Edad Media se construyen otros autómatas, como son el *Hombre de hierro* de Alberto Magno (1204-1282), la *Cabeza parlante* de Roger Bacon (1214-1294) o el *Gallo de Estrasburgo* (1352). Este último es el ejemplo más importante al ser el autómata más antiguo que se conserva, funcionó desde 1352 hasta 1789, formaba parte del reloj de la catedral de Estrasburgo y al dar las horas movía el pico y las alas.

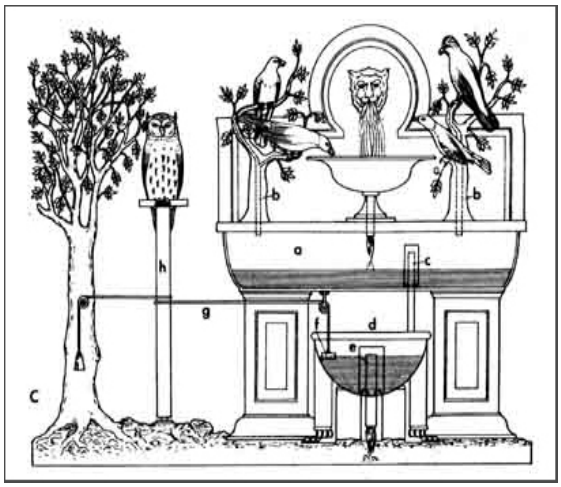


Figura 1. Pajaros de Herón

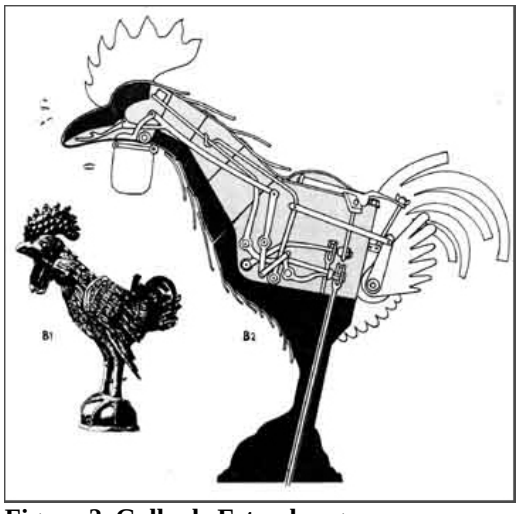


Figura 2. Gallo de Estrasburgo

Algunos de las figuras más relevantes del Renacimiento se interesaron durante los siglos XV y XVI por los ingenios desarrollados por los griegos. Leonardo Da Vinci (1452-1519) construyo el *León Mecánico* para el rey Luis XII de Francia; el león se abría el pecho con su garra y mostraba el escudo de armas del rey. En España, Juanelo Turriano construye el *Hombre de palo* para el emperador Carlos V<, este autómata tenía forma de monje y podía andar y mover los brazos, la cabeza, los ojos y la boca.

Durante los siglos XVII y XVIII se crearon autómatas que comienzan a adquirir alguna de las características de los robots actuales. Creados en su mayoría por artesanos del gremio de la relojería y con fines lúdicos para entretener a las gentes de la corte y servir de atracción a las ferias. Un artesano llamado Camus (1576-1626) construyó para Luis XIV un coche de caballos en miniatura con sus lacayos y una dama. Las figuras de la miniatura de Camus tenían capacidad para moverse.

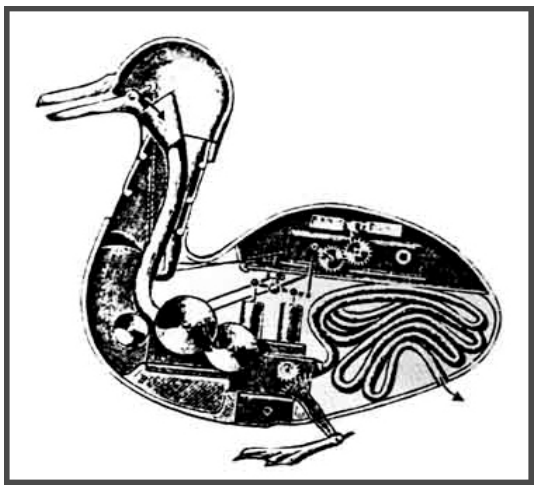


Figura 3. Pato de Vaucanson

Jacques de Vaucanson (1709-1782) construyó, entre otros ingenios mecánicos, un increíble pato mecánico que fue la admiración de toda Europa. Según palabras de Sir David Brewster en 1868, el pato es *"la pieza mecánica más maravillosa que se haya hecho"*. El pato podía comer tragar y digerir y evacuar la comida. Además podía beber, chapotear y graznar.

Un relojero suizo llamado Pierre Jaquet Droz (1721-1790),con la ayuda de sus hijos Henri-Louis y Jaquet, construyó diversos muñecos capaces de escribir, dibujar y tocar diversas melodías en un órgano. Estos muñecos todavía se conservan en el museo de arte e Historia de Neuchâtel, Suiza. Henry Maillardet construyó una muñeca capaz de dibujar.

Figura 4. Autómata de Jaquet Droz

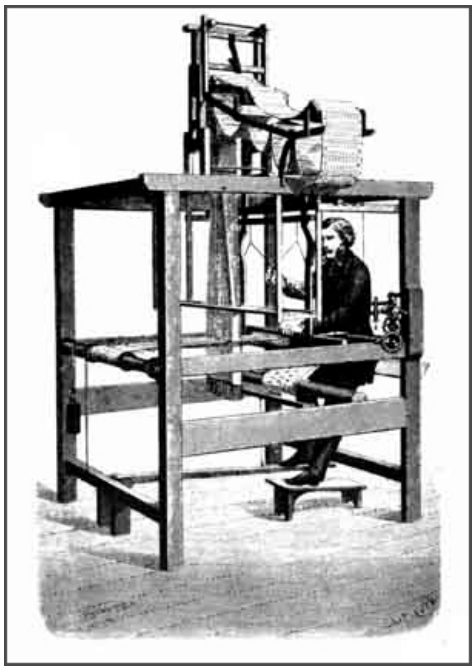


Figura 5. Telar de Jacquard

A finales del siglo XVIII y principios XIX se desarrollaron ingenios útiles para la industria textil, lo que permitió la utilización de dispositivos automáticos en la producción industrial, dando paso a la automatización industrial. Algunos de estos inventos son la *hiladora giratoria de Hargreaves* (1770), la *hiladora mecánica de Crompton* (1779), el *telar mecánico de Cartwright* (1785) y el *telar de Jacquard* (1801). Este último utilizaba las tarjetas perforadas como soporte de un programa para las acciones de la máquina, es decir, se podía definir el tipo de tejido a realizar. Estas máquinas constituyeron los primeros precedentes históricos de las máquinas de control numérico.

La primera vez que se utiliza el término **robot** es en 1921 cuando un escritor checo llamado Karel Capek publica su obra *Rossum Universal Robots*. El origen del título viene de la palabra eslava "*robota*" que significa trabajo forzado. En la obra aparecen máquinas androides que sirven a sus jefes humanos, hasta que finalmente se rebelan acabando con toda forma de vida humana excepto sus creadores, con la esperanza de que éstos les pudieran enseñar a reproducirse.

El término volvió a aparecer en la ciencia ficción, motivo por el cual no cayó en desuso. En 1926, Thea von Harbou escribió *Metrópoli*. Novela en la que un androide llamado María manipula una sociedad obrera superindustrializada.

En 1932 aparece el primer robot de juguete, es bautizado como *Lilliput*, funcionaba bajo los principios de relojería, fabricado en acero estampado y basado en varios juguetes alemanes y estadounidenses que se fabricaban antes de la guerra.

La primera vez que se utiliza la palabra robótica es cuando el gran escritor Isaac Asimov escribe *Runaround* (1942). En octubre de 1945 escribió en la revista *Galaxy Science Fiction* una historia en la que enunció las Tres Leyes de la Robótica:

I. *Un robot no puede hacer daño a un ser humano o, por inacción, permitir que un ser humano sufra daño.*
II. *Un robot debe obedecer las órdenes dadas por los seres humanos, excepto si estas órdenes entrasen en conflicto con la Primera Ley.*
III. *Un robot debe proteger su propia existencia en la medida en que ésta protección no entre en conflicto con la Primera o la Segunda Ley.*

Otro Principio, también propuesto por Asimov, es la Ley Cero o "Zeroth Law":

"Un robot no puede hacer daño a la Humanidad o, por inacción, permitir que la Humanidad sufra daño"

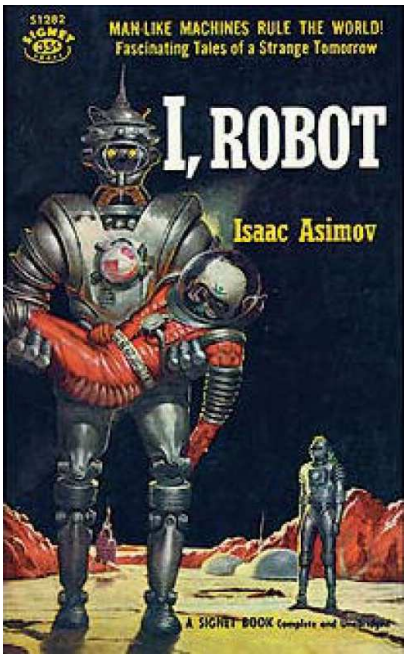


Figura 6. Portada de la novela *Yo, Robot* de Isaac Asimov

3.2 DESARROLLO DE LA ROBÓTICA

Tras los primeros autómatas citados en el apartado anterior hay que esperar hasta 1948, cuando R.C. Goertz de *Argonne National Laboratory* desarrolló el primer *telemanipulador* con el propósito de manipular materiales radiactivos sin peligro para el operador. Consistía en un dispositivo mecánico maestro-esclavo.

En 1954 Goertz sustituye la transmisión mecánica por tecnología electrónica y de servocontrol desarrollando el primer *telemanipulador con servocontrol bilateral*.

En 1958 Ralph Mosher desarrolló el dispositivo conocido como *Handy-Man*, consistía en dos brazos mecánicos teleoperados por un maestro. Las industrias submarina y espacial empiezan a interesarse por estos telemanipuladores.

Sin embargo, estos telemanipuladores no han evolucionado de la misma forma que los robots. Ésto es debido a que para controlar el manipulador es necesario un operario que maneje de forma continua el *maestro*. Para resolver este problema se empieza a pensar en la sustitución del operario por un programa de ordenador. De esta idea nace el concepto de robot.

La primera patente de un dispositivo robótico fue solicitada en 1954 por el inventor de origen británico C.W. Kenward. Sin embargo, el inventor e ingeniero norteamericano George C. Devol estableció las bases de la robótica moderna al idear un dispositivo de transferencia de artículos programada patentándola en 1961. Devol dio a conocer esta idea a Joseph F. Engelberger, director de ingeniería de la división aeroespacial de la empresa *Manning Maxwel y Moore*. Juntos empezaron a trabajar con vistas a la utilización de sus máquinas en la industria y así crearon la empresa bautizada como *Consolidated Controls Corporation*, que más tarde fue rebautizada como *Unimation* (Universal Automation).

En 1960, Unimation instala su primera máquina, llamada *Unimate*, en la fábrica de General Motors en Trenton, para una aplicación de fundición por inyección. Otras empresas importantes como AMF empiezan la creación de máquinas similares.

Figura 7. Primer robot instalado en la industria americana

Con la visita de Engelberger a Japón en 1968, en donde firmó contratos con Kawasaki, empieza a crecer la robótica en Japón superando en breve tiempo a la robótica norteamericana gracias al apoyo de grandes empresas como *Nissan*.

Figura 8. Robot IRb6 de AESA

La robótica no llegó a Europa hasta el año 1973. Fue entonces cuando la empresa sueca AESA construyó el primer robot con accionamiento totalmente eléctrico denominado, *IRb6*.

En la década de los 70 del pasado siglo, los microprocesadores de Intel junto con inventos de otras grandes compañías impulsaron fuertemente la tecnología y la robótica. Sin embargo, fueron las novelas y películas de ciencia ficción como las de George Lucas, las que dieron a conocer los robots al público general.

Figura 9. Robot R2D2 de la saga Star Wars

Las primeras configuraciones utilizadas en el desarrollo de robots fueron las denominadas configuraciones esféricas y antropomórficas. En 1982, el profesor Makino de la *Universidad Yamanashi* de Japón, desarrolla el robot de tipo SCARA (*Selective Compliance Assembly Robot Arm*) cuyas principales características son pocos grados de libertad (3 o 4), coste limitado y orientado hacia el ensamblaje de piezas.

En tan sólo 30 años, la evolución de los robots industriales ha sido vertiginosa. Actualmente, el uso de robots industriales se encuentra altamente masificado, tomando posiciones en casi todas las áreas productivas y tipos de industria. Los robots sustituyen al hombre en tareas repetitivas, hostiles y laboriosas y son capaces de adaptarse fácilmente a cambios en la producción.

Aunque la mayor parte de los robots actuales se utilizan en la industria, existen robots dedicados a otras aplicaciones de carácter no industrial, como por ejemplo, los robots espaciales, los robots utilizados en aplicaciones médicas, robots militares, robots para aplicaciones submarinas y subterráneas, y un largo etcétera.

En las siguientes imágenes se pueden observar dos de los robots de última generación de Staubli, el *TP80 Fast Picker*, robot de empaquetado de 4 ejes y el *RX170 hsm Machining*, robot de 6 ejes.

Figura 10. TP80 Fast Picker

Figura 11. RX170 hsm Machining robot

3.3 DEFINICIÓN DE ROBOT INDUSTRIAL

Encontrar una definición formal de lo que es un robot industrial puede ser una tarea complicada. Existe, de inicio y entre otras dificultades, una gran dispersión conceptual acerca del término robot: el concepto de robot es diferente para el mercado japonés y para el mercado occidental. Para el mercado japonés un robot es cualquier dispositivo mecánico dotado de articulaciones destinado a la manipulación. Sin embargo, el mercado occidental es más restrictivo y exige una mayor complejidad en la definición, haciendo especial hincapié en el control.

Dentro del mercado occidental tampoco existe un acuerdo para dar una definición formal, pero si existe una idea común sobre lo que es un robot industrial. También hay que tener en cuenta que el paso del tiempo y la evolución propia de la robótica obligan a actualizar las definiciones.

Según el *diccionario Webster*:

> *Un robot es un dispositivo Automático que efectúa funciones ordinariamente asignadas a los seres humanos*.

Según la RIA (*Robot Industry Association*):

> *Un robot industrial es un manipulador reprogramable multifuncional diseñado para mover materiales, piezas, herramientas o artefactos especiales, mediante movimientos variables programados, para la ejecución de tareas potencialmente muy diversas*.

Según la ISO (*International Organization for Standardization*)**:**

> *Un robot es un manipulador reprogramable multifuncional que posee varios grados de libertad, con capacidad de manipular materias, piezas, herramientas o artefactos especiales según trayectorias variables programadas para la ejecución de tareas diversas*.

AFNOR (*Association Française de Normalisation*) establece una definición más completa, especificando previamente qué se entiende por *manipulador*:

> *Un manipulador es un mecanismo formado generalmente por elementos en serie, articulados entre sí, diseñado para el agarre y desplazamiento de objetos. Es multifuncional y puede ser gobernado directamente por un operador humano o mediante un dispositivo lógico.*
>
> *Un robot es un manipulador automático servo-controlado, reprogramable, polivalente, con capacidad de posicionar y orientar piezas, útiles o dispositivos especiales, siguiendo trayectoria variables reprogramables, para la ejecución de tareas variadas. Normalmente tiene la forma de uno o varios brazos terminados en una muñeca. Su unidad de control incluye un dispositivo de memoria y ocasionalmente de percepción del entorno. Generalmente su uso es el de realizar una tarea de manera cíclica, pudiéndose adaptar a otra sin cambios permanentes en su material.*

Por ultimo, la IFR (*Internationa Federation of Robotics*) define robot industrial de manipulación y lo distingue de otros robots:

> *Por robot industrial de manipulación se entiende una maquina de manipulación automática, multifuncional y reprogramable, con tres o más ejes que pueden posicionar y orientar materias, piezas, herramientas o dispositivos especiales para la ejecución de trabajos diversos en las diferentes etapas de la producción industrial, ya sea en una posición fija o en movimiento.*
>
> De la definición se entiende que la reprogramabilidad y la multifunción se consiguen sin modificaciones físicas del robot.

Hay que añadir en este apartado la diferencia entre la robótica industrial y la robótica de servicios. La robótica industrial de manipulación nace de exigencias prácticas en la producción, la robótica es un elemento importante de la automatización flexible, encaminada a la reducción de costes. En cambio, un robot de servicios es un dispositivo electromecánico, móvil o estacionario, dotado generalmente de uno o varios brazos mecánicos, controlados por ordenador y diseñados para la realización de tareas no industriales de servicio. Algunos ejemplos de robots de servicios pueden ser los robots diseñados para la medicina, educación, aplicaciones espaciales o submarinas.

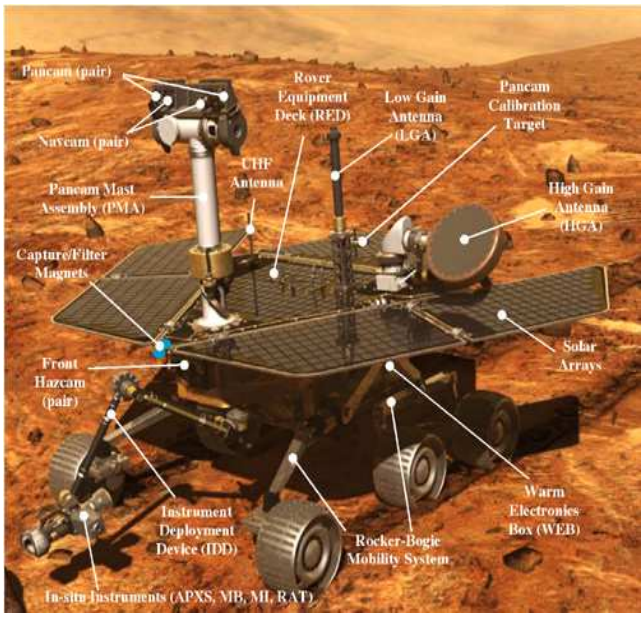


Figura 12. Robot espacial rover Spirit.

3.4 CLASIFICACIÓN DE LOS ROBOTS INDUSTRIALES

Aún cuando existen variadas formas de clasificar los robots por las distintas Agencias de robótica o por los autores especializados, ya sea teniendo en cuenta la generación a la que pertenece el robot, ya sea atendiendo a los movimientos que puede realizar, se adoptará una clasificación que atiende tanto a los movimientos que realizan como a la forma en que se controlan.

A) Manipuladores secuenciales

- Son capaces de llevar a cabo movimientos definidos de forma discreta entre situaciones predefinidas.
- Utilizan microrruptores, finales de carrera, etc.
- Su funcionamiento es controlado por autómatas programables.

B) Robots con control numérico (NC robots)

- Realizan trayectorias continuas definidas por guiado y mediante un programa en lenguaje simbólico o únicamente por un programa.
- Están provistos de un Subsistema que interpreta instrucciones codificadas en cierto lenguaje de alto nivel.

C) Robots "inteligentes"

- Se caracterizan porque analizan el estado de su entorno, toman decisiones y generan sus propios planes de acción.
- Poseen un Subsistema de control con cierta complejidad, debido a que integra técnicas de reconocimiento de formas y de inteligencia artificial.
- Disponen de un sistema de percepción constituido por muchos sensores de varios tipos
- Es la generación de robots que se está desarrollando e investigando actualmente.

Además de estos robots cuyo campo de actuación es el sector industrial, hay otro amplio número de dispositivos robóticos que posibilitan la realización de tareas en ambientes de riesgo o con mayor comodidad que con un procedimiento manual o que evitan operaciones excesivamente repetitivas. Se agrupan en la denominada robótica de servicios o de intervención.

Se encuentran en este campo telemanipuladores utilizados en ambientes de riesgo o en cirugía; robots móviles autónomos o robots móviles teleoperados como los usados en la exploración marina o en la desactivación de explosivos

3.5 SIMULADORES ROBÓTICOS

Como se ha comentado anteriormente, existen multitud de simuladores robóticos. La finalidad de un simulador robótico es permitir la creación de aplicaciones o tareas robóticas sin depender físicamente de una máquina o robot. Esto permite ahorrar tiempo y dinero, y facilita el aprendizaje sin realizar una fuerte inversión monetaria. El término simulador robótico puede ser referido a muchas aplicaciones robóticas simuladas.

Una de las aplicaciones de los simuladores robóticos es la creación de un entorno virtual modelado y renderizado en 3D. Este tipo de software robótico tiene la capacidad, mediante un robot virtual, de emular los movimientos de un robot industrial en un entorno real. Para aumentar el realismo en los movimientos del robot virtual algunos de estos simuladores poseen un complejo motor de físicas.

El uso de simuladores robóticos está altamente recomendado. Una de las ventajas de un simulador robótico es que permite la programación y depuración de la tarea sin necesidad de la utilización del robot real. Esto ofrece facilidades a la hora de realizar el trabajo debido a que la programación de la tarea puede hacerse de forma *off-line* y, por tanto, sólo será necesario el uso del robot real para probar el código, de la versión final y sin fallos, realizado con el simulador. El éxito de la simulación será mayor cuanto más parecido tenga el entorno real con el entorno virtual. Hay que tener en cuenta, que no siempre será posible una simulación completa y exacta de lo que sucedería en el entorno real. La simulación de tareas basadas en sensores tendrá una mayor complejidad puesto que los movimientos del robot dependen de la lectura instantánea de los sensores del entorno real.

No siempre se utilizan los simuladores robóticos con la intención de programar tareas que posteriormente serán llevadas a un entorno real. En múltiples ocasiones la intención de su utilización es simplemente educativa. Existen por tanto, simuladores diseñados para tal fin. En los simuladores robóticos con finalidad educativa normalmente es más importante el manejo y sencillez de uso que una representación exacta del entorno real. De esta manera el usuario podrá comprender mejor las materias robóticas.

Por otro lado cabe destacar que los avances computacionales han favorecido la existencia de simuladores robóticos. Primero, por la aparición de las GPU (*Graphics Processing Unit*) y los primeros motores gráficos 3D, a finales de los años 80. Posteriormente por el aumento de las capacidades gráficas y de procesamiento actuales que permiten mover entornos gráficos 3D con multitud de componentes gráficos como efectos, luces, texturas, polígonos, etc. y todo ello con gran fluidez.

Los simuladores robóticos actuales tratan de alcanzar ciertas características:

- Creación rápida de robots. A menudo los propios simuladores permiten la creación del robot dentro del propio programa. En otras ocasiones se realiza mediante la utilización de herramientas externas.
- Renderizado 3D realístico.
- Motores de físicas para dar realismo a los movimientos de la escena.

- Lenguajes de programación basados en lenguajes de propósito general para la programación de la tarea robótica.

A continuación revisamos algunos de los simuladores robóticos más importantes que hemos dividido en 2 grupos. Por un lado podemos encontrar los *simuladores de robots móviles*. La principal característica de estos simuladores es que están enfocados hacia el aprendizaje de la dinámica de robots móviles, siendo una parte importante de la simulación el comportamiento del robot con el entorno mediante la utilización de sensores de visión, de distancia o de contactos.

Por otro lado, podemos encontrar *simuladores robóticos industriales*. Estos simuladores son los más interesantes para nuestro objetivo debido a que el propósito del proyecto es la actualización de uno de estos simuladores. Estos últimos están enfocados hacia la simulación de un entorno industrial, en el que se utiliza el robot para satisfacer las exigencias prácticas de la producción. Se trata de simular escenarios robóticos en los cuales el robot interactúe con el entorno, de manera lo más cercana posible a la realidad, y enfocados al aprendizaje de la robótica industrial o a permitir la programación de la tarea sin disponer del robot.

3.5.1 Simuladores de robots móviles

Gazebo:

Es un completo simulador 3D multirobot de código abierto. Empezó a desarrollarse en el año 2002 en la universidad de Southern California. **Gazebo** esta capacitado para la simulación de robots articulados en entornos complejos y realísticos aunque está orientado hacia robots móviles y androides (esto es, no industriales).

Sus principales características son:

- Capacidad de simulación de sensores robóticos como láseres, sonar, cámaras, etc.
- Incluye modelados de algunos robots comúnmente usados, como ejemplo, incluye modelado de los robots PR2, Pioneer2DX, Pioneer2AT y SegwayRMP.
- Simulación realística de físicas, el robot puede arrojar y recoger objetos, así como interactuar con el entorno.
- El usuario puede desarrollar sus propios robots y sensores, cargándolos dinámicamente en tiempo de ejecución.
- Apariencia realística obtenida a través de programas de modelado en 3D.

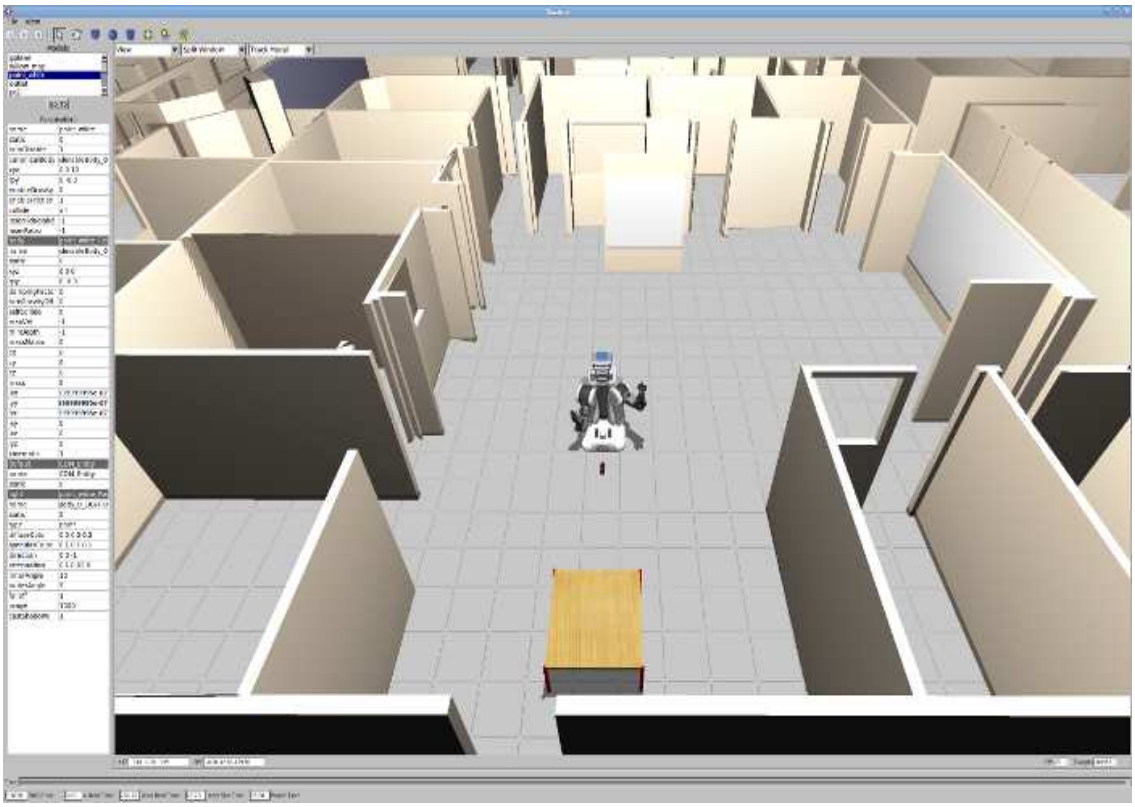


Figura 13. Captura de pantalla del simulador Gazebo

Simbad:

Simulador robótico bastante sencillo, escrito en Java con fines didácticos. Enfocado hacia simulaciones de robots móviles e inteligentes. Permite el estudio de máquinas inteligentes, inteligencia artificial, y conocer el funcionamiento de robots autónomos. No tiene la intención de simular entornos del mundo real.

Simbad permite al usuario la recreación, simulación y desarrollo del controlador de su propio robot, modificar el entorno y la utilización de sensores. Características:

- Entorno 3D para la visualización y utilización de sensores.
- Permite simulación multirobótica.
- Sensores de visión.
- Sensores de distancia: sonar e infrarrojos.
- Sensores de contacto.
- Interfaz de usuario para el control.

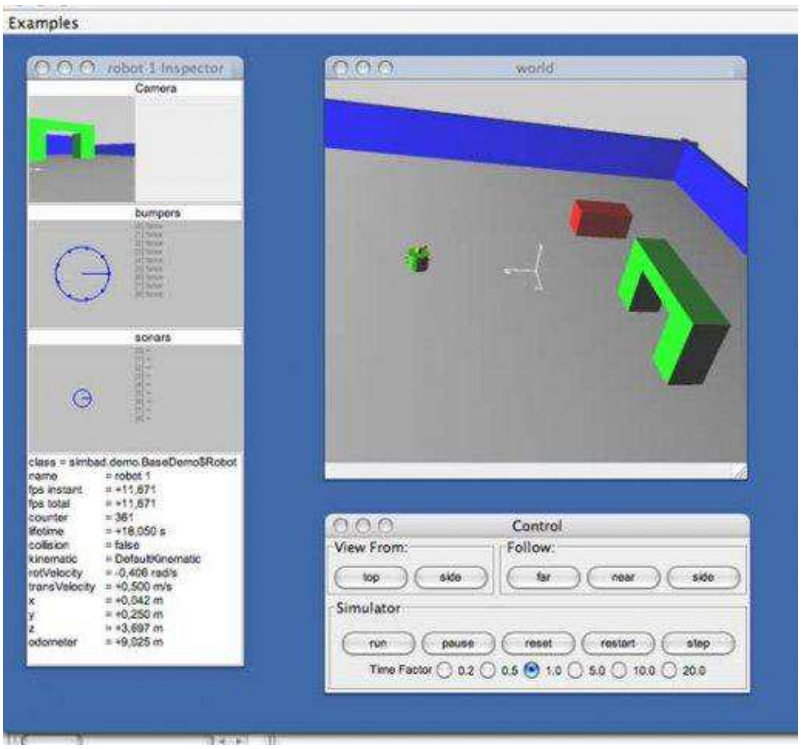


Figura 14. Captura de pantalla del simulador Simbad

OpenHRP3 (Open Architecture Human-centered Robotics Platform):

Es una plataforma de código abierto para la simulación y desarrollo de robots móviles con forma humana. Permite al usuario inspeccionar un modelado robótico y su control de programa mediante simulación de la dinámica y movimientos del robot.

Sus principales características son:

- Simulación de físicas, cinemática y dinámica.
- Simulación de sensores.
- Representación de gráficas tiempo par de las articulaciones.

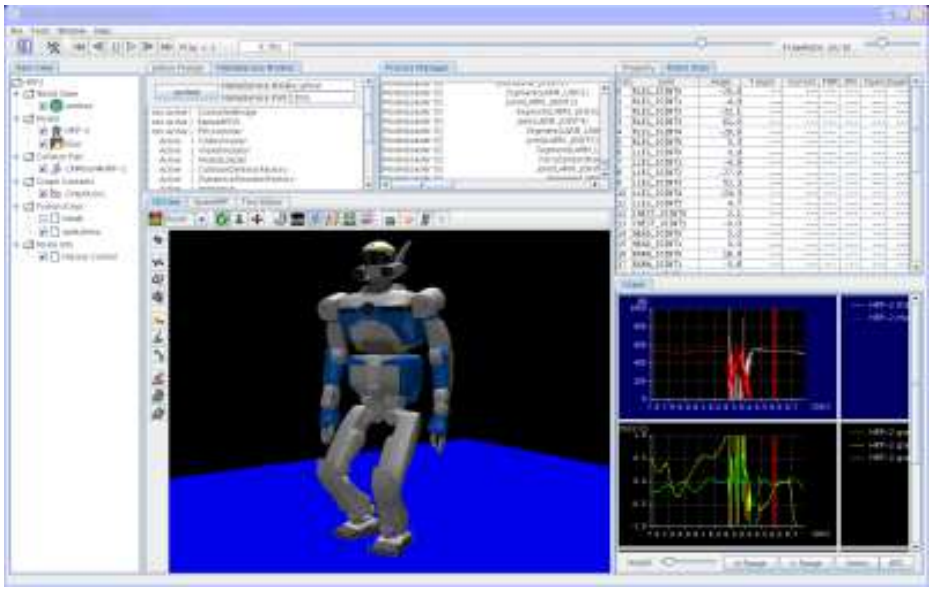


Figura 15. Captura de la plataforma OpenHRP3

Webots:

Es un completo simulador profesional diseñado con propósitos educativos y orientado principalmente hacia la simulación de robots móviles. El proyecto comenzó en el año 1996 en el Instituto Tecnológico Federal de Suiza (EPFL) en Lausana.

Webots utiliza el motor ODE para la detección de colisiones, y la simulación de físicas y dinámicas. El programa cuenta con una colección de robots que pueden ser modificados libremente, además permite la creación de nuevos modelos pudiéndose detallar las características gráficas y físicas. Cuenta con un conjunto de sensores que son comúnmente utilizados en robótica: sensores de contacto, de proximidad, de luz, acelerómetros, cámaras, etc.

Los programas pueden escribirse en *C, C++, Java, Python y Matlab*. Como curiosidad, se debe destacar que **Webots** es utilizado en varios concursos de programación como la RoboCup.

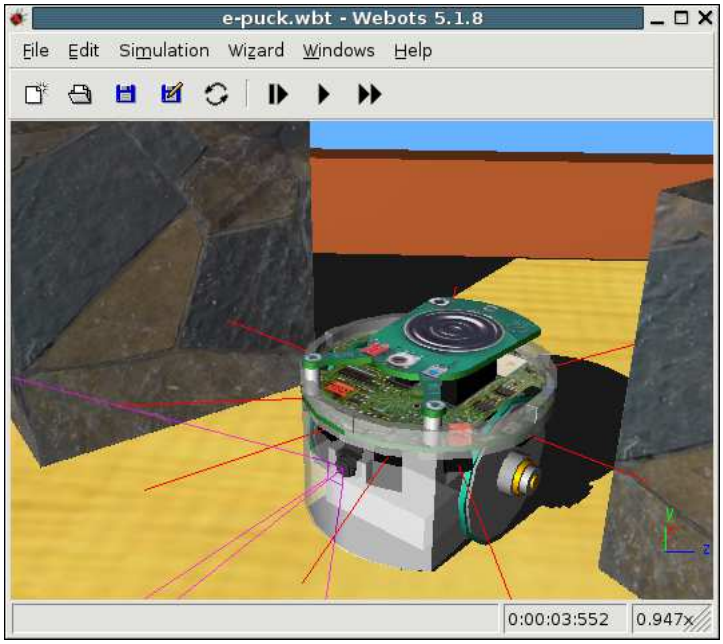


Figura 16. Simulador Webots

3.5.2 Simuladores de robots industriales

RoKiSim:

Es un software educativo y libre para realizar simulaciones sencillas en 3D de robots de 6 ejes del tipo PUMA. Incluye modelados de algunos robots (*ABB IRB 120*, *ABB IRB 140*, *ABB IRB 1600/1.45*, *CRC A465*, *Fanuc LR Mate 200iC*, *KUKA KR 5 sixx R650*, *KUKA KR 1000 TITAN* y *Motoman UP50N*). Es relativamente sencillo añadir nuevos modelados. Trae simulaciones para observar los distintos tipos de singularidades y es posible importar otros

objetos geométricos a la escena. Es posible cambiar el idioma de los menús entre unos cuantos disponibles (Inglés, Español, Francés y Catalán).

Figura 17. Captura del simulador RoKiSim

Virtual Robot Experimentation Platform:

Es un simulador robótico comercial 3D, con un entorno de desarrollo integrado donde es posible la asignación de scripts a los elementos de la escena, dichos scripts pueden ejecutarse de manera simultánea para la realización de tareas multirobóticas. Con este software es posible la simulación de otros sistemas robóticos como sensores y mecanismos.

Otras de sus características son:

- Permite controlar un robot real de forma remota.
- Dos motores de físicas para simular físicas reales e interacciones con objetos de la escena.
- Cálculos de la cinemática inversa.
- Es posible la simulación de fluidos como aire o agua.
- Detector de colisiones.
- Simulación de sensores de proximidad o de visión.
- Permite la importación y exportación de modelados en varios formatos CAD.
- Explorador de modelados para componer escenarios.

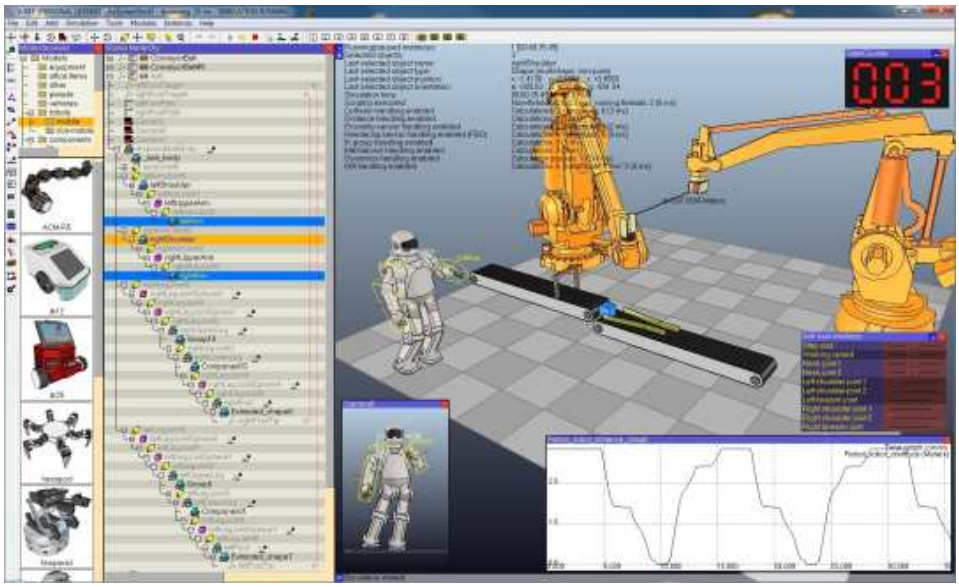


Figura 18. Captura de pantalla del simulador V-rep

RoboLogix:

Es un software diseñado para emular aplicaciones robóticas del mundo real. Utilizando un robot industrial de 5 ejes, **RoboLogix** permite mostrar, correr, probar y depurar programas escritos para simular una amplia variedad de aplicaciones prácticas. Algunas de estas aplicaciones pueden ser las tareas de paletizado y montaje, aunque el programa permite la creación de nuestros propios entornos robóticos. El usuario puede ejecutar y examinar los programas robóticos y los algoritmos de control mientras se visualizan las velocidades, posiciones y las aceleraciones en las articulaciones.

Esta diseñado para estudiantes y ofrece una buena simulación (basada en ingeniería) a un precio aceptable. El programa verifica rangos de alcance y detecta colisiones, de este modo, la simulación es más fiel a la realidad. Con **RoboLogix** se pueden crear nuestros propios programas y modificar elementos del entorno. Cuenta con varios sensores (incluidas cámaras de video) que permiten obtener las posiciones del robot, además de incorporar ejemplos y demos para facilitar la comprensión por parte del usuario.

Figura 19. Simulador RoboLogix

Visual Components:

Es un simulador comercial de robots industriales. Su catálogo de robots permite la utilización de robots industriales de 15 marcas diferentes. Todos los modelos del catálogo son perfectamente funcionales y vienen preconfigurados para permitir la correcta visualización del área de trabajo y poder trabajar con diferentes robots y configuraciones.

Una de las ventajas de este simulador es que permite la creación de entornos robóticos de una manera rápida y sencilla. La creación del escenario se puede realizar arrastrando y soltando objetos.

La programación del robot se realiza también de una forma bastante sencilla con un editor basado en iconos. El simulador cuenta con detector de colisiones, comprobación del rango de alcance y permite escoger la herramienta del robot en función de la tarea robótica a realizar.

Figura 20. Imagen del Software Visual Components

WorkSpace LT:

Es un simulador de robótica y automática de corte comercial muy completo. **WorkSpace** cuenta con varias versiones, una dirigida a la industria y otra versión para el mercado educativo.

La versión dirigida a la industria está diseñada para facilitar la tarea del programador, por tanto, cuenta con ciertos beneficios inherentes a la programación *off-line* como son la reducción de costes, reducción del tiempo de parada de la producción, mejora de la flexibilidad, acelera el tiempo de diseño del entorno, previene de daños en el equipo y permite la visualización de la tarea.

La versión desarrollada para el mercado educativo es más interesante para nosotros debido a que permite abordar el aprendizaje de los conceptos

robóticos y otros equipos diseñados para la automática. Ofrece programación *off-line* completa y la posibilidad de simular entornos de trabajo complejos con múltiples robots y dispositivos automáticos. Presenta las siguientes características:

- Permite la creación de sólidos mediante herramientas de CAD 3D, así como, su manipulación y modificación de sus características, pudiéndose importar los módelos en archivos con formatos CAD.
- Creación de nuestros propios robots y mecanismos. El simulador dispone de una librería de robots.
- Preciso sistema de cinemáticas para robots y automatismos.
- Soporta la simulación estándar E/S entre robots y mecanismos.
- Alta precisión en estudios del rango de alcance y tiempos de ciclo del robot.
- Permite detectar colisiones.
- Soporta los lenguajes de programación nativos para robots de diferentes marcas: ABB (Arla , Rapid), Adept (VPlus), Esched (ACL), Fanuc (Karol, TP), JIRA (Strolic), Kawasaki (AS), Mitsubishi (MMB), Motoman (Inform1, Inform2), Nachi (Slim), Unimation (Val1, Val2).
- Posibilidad de generar videos de la simulación.

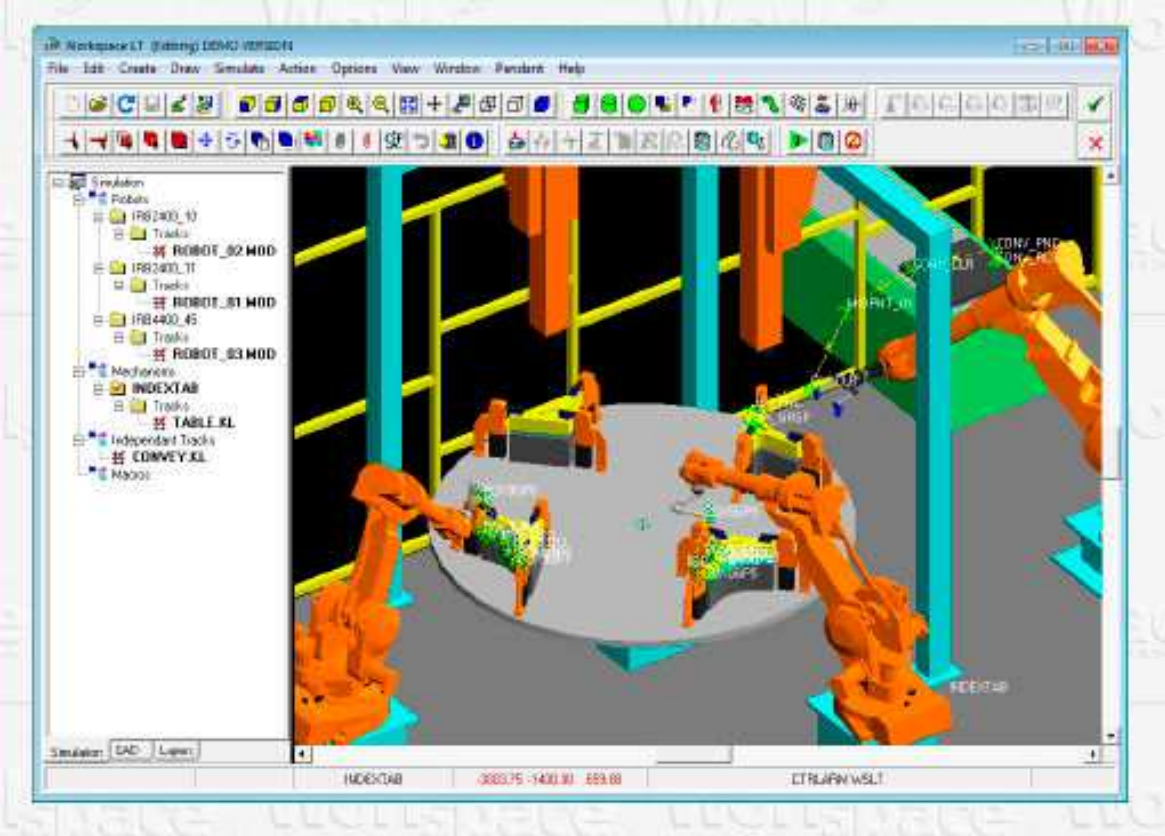


Figura 21. Captura de pantalla del simulador WorkSpace LT

3.6 ROBOTSCENE COMO PUNTO DE PARTIDA

Como se ha comentado anteriormente, el objetivo primordial del proyecto es la correcta mejora y actualización de **RobotScene** de manera que el programa sea capaz de ejecutar y simular tareas en las que intervengan varios robots que, además, puedan comunicarse entre sí. Es necesario conocer el estado del programa en el momento en que se inició la realización de este proyecto, de esta manera conoceremos el punto de partida inicial del proyecto y podremos observar el progreso realizado.

Repasaremos las características principales de la versión mono-robot de **RobotScene**, e indagaremos sobre el análisis de las posibles soluciones, por cual de estas soluciones se decidió optar y por qué razón.

Figura 22. Logotipo del simulador RobotScene

3.6.1 Características de RobotScene (versión mono-robot)

RobotScene es un completo simulador robótico gratuito, diseñado principalmente para facilitar el aprendizaje de la robótica a estudiantes de ingeniería. Posee tres módulos diferenciados para la creación de objetos, de robots y de escenarios robóticos de forma intuitiva, rápida y visual.

RobotScene permite crear sin apenas esfuerzo cualquier robot industrial que se desee, sin limitaciones en cuanto a su morfología o número de articulaciones, pudiendo incluir dicho robot posteriormente en un escenario robótico en 3D para la simulación de tareas industriales reales.

Características principales de la versión mono-robot de **RobotScene**:

- Aplicación autocontenida que no precisa de otros programas ni compilaciones para crear sus propios robots.
- Uso de API gráfica OpenGL para la representación gráfica en 3D de objetos, robots y escenarios. De esta forma se incorpora la posibilidad de utilización de antialiasing, profundidad de color, proyección de sombras y captura de imágenes y videos.
- Detección de colisiones entre robot y elementos del escenario.
- Constructor de objetos 3D con una interfaz intuitiva al estilo de 3DStudio y que permite la creación de objetos con formas complejas

mediante extrusión. Posee una librería de materiales para añadir texturas a los objetos creados.

- Construcción de robots de manera sencilla mediante los parámetros de Denavit y Hartenberg, permite la creación de herramientas propias y de definir configuraciones específicas de los robots.
- Implementación del modelo inverso de los robots mediante scripts de programación con sintaxis parecida a Pascal y que permite el tratamiento de singularidades.
- Constructor de escenarios que permite la configuración de escenas y tareas robóticas. Explotación del escenario mediante guiado, registro de trayectorias y programa robótico. Se permite la inclusión de varios robots en el escenario, pero sólo es posible la explotación de uno de ellos.

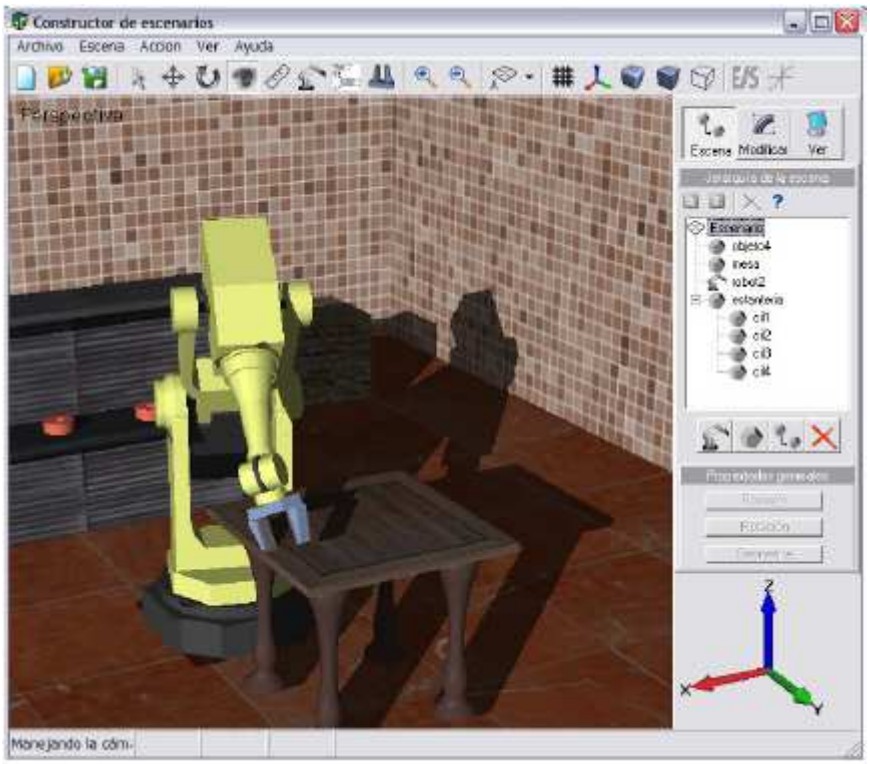


Figura 23. RobotScene (mono-robot). Constructor de escenarios.

3.6.2 Análisis de la solución seleccionada en RobotScene (versión mono-robot)

Es el momento ahora de analizar las diferentes soluciones consideradas al inicio de la versión mono-robot de **RobotScene**. Si se desea dotar a **RobotScene** con la capacidad para ejecutar tareas en las que intervengan varios robots, es necesario conocer las diferentes alternativas pensadas y adoptadas en su diseño y desarrollo originales.

Cuando se aborda el desarrollo de una actualización de un programa informático hay que tener en cuenta las soluciones asumidas en la elaboración de dicho programa. De este modo, se puede comprender por qué ciertas partes de la aplicación están desarrolladas de una determinada manera, lo que facilita el correcto entendimiento de las líneas de código y permite orientar el trabajo a realizar para alcanzar el objetivo principal.

Finalmente, se puede considerar que si se conoce cómo está hecho el programa se reconocen a su vez las limitaciones dependientes de una solución concreta, lo que obliga en unas ocasiones a amoldarse a las decisiones tomadas previamente y, en otras ocasiones, a modificar dicha solución para la consecución del nuevo objetivo.

Las primeras decisiones importantes que se adoptaron en el diseño de la versión mono-robot de **RobotScene** fueron las referidas al lenguaje de programación empleado, al compilador de dicho lenguaje y a la plataforma sobre la cual se ejecutaría la aplicación.

En el primer momento de desarrollo, se pensó en que el simulador fuera capaz de construir escenarios robóticos con varios robots. Para realizar esta tarea es necesario que el lenguaje de programación pueda ejecutar varios flujos de código de forma simultánea y en tiempo real; en otras palabras, que el programa pueda ejecutar diferentes partes de programa de forma concurrente. En resumidas cuentas, sería necesario que el programa ejecutara un flujo de programa por cada robot (más otro flujo de programa para controlar la parte general del programa).

Con la idea de que el lenguaje de programación permitiera dicha concurrencia se estudió la posibilidad de escribir el programa en ADA o Java, como primeras opciones. Como segundas opciones se pensó en la utilización de lenguajes como C++ o Delphi debido a que éstos también permiten la programación concurrente aunque de una manera menos sencilla.

De estas dos primeras opciones se descartó el uso de Java debido a que no genera archivos ejecutables y se debe ejecutar sobre una máquina virtual. La utilización de una máquina virtual presenta una ventaja importante puesto que permite ejecutar la aplicación en las plataformas que tengan portada la máquina virtual. Sin embargo, la máquina virtual presenta también una gran desventaja puesto que su utilización consume recursos lo que disminuye la velocidad de la aplicación. Esta gran desventaja obligó a descartar el lenguaje Java porque la velocidad de la aplicación es crucial en aplicaciones que requieran un uso intensivo de gráficos en 3 dimensiones.

ADA es un lenguaje multipropósito, orientado a objetos y concurrente, pudiendo llegar desde la facilidad de Pascal hasta la flexibilidad de C++.

Una de las partes importantes a la hora de desarrollar una aplicación es que el usuario se pueda comunicar de forma intuitiva con la aplicación, por tanto, desde un primer momento se sabía que iba a ser necesaria la creación de una interfaz gráfica de usuario (GUI). En este momento se observó que no existen

entornos integrados de desarrollo para ADA que permitan crear de forma gráfica la interfaz independientemente de la plataforma como puede ser el caso de Visual C++ de Microsoft mediante su MFC (*Microsoft Foundation Class*) o de Borland con Delphi mediante la VCL (Visual Component Library). La utilización de estas librerías evita al programador de la aplicación tener que enfrentarse con aspectos dependientes de la plataforma en la que se está programando. Así, por ejemplo, en Windows, se evita que el programador tenga que tratar directamente con el API de win32, la cual se encapsula debidamente y se le ofrece al programador una estructura orientada a objetos y un tratamiento de los eventos que generan dichos objetos.

Windows ADA permite el uso directo del API Win32, pero esta opción se descarta inmediatamente debido, principalmente, a que es necesario poseer ciertos conocimientos del API de Windows; así, el código resultante es muy complejo y el tiempo necesario para generar dichos gráficos no compensa en absoluto. Además, el uso directo del API de Windows elimina la opción de poder portar la aplicación a otras plataformas.

Por todo lo anterior, para utilizar ADA, aparecía como necesario el uso de librerías capaces de encapsular debidamente las funciones propias del sistema operativo, como pueden ser las librerías GTK+.

GTK+ es un grupo importante de bibliotecas o rutinas para desarrollar interfaces gráficas de usuario (GUI). Es software libre pero presenta un inconveniente bastante importante porque la creación de la interfaz gráfica debe hacerse mediante texto, lo que aumenta en gran medida la complejidad al no tener una referencia visual, y obliga, a su vez, a emplear tiempo y esfuerzo ante cualquier mínimo cambio que se quiera realizar en el apartado gráfico.

Para evitar el uso de texto en la creación de la interfaz gráfica, se nos ofrece la posibilidad de usar Glade, herramienta de desarrollo visual de interfaces gráficas, que permite diseñar la interfaz de forma gráfica similar a como se haría con un Visual Basic y genera el código correspondiente.

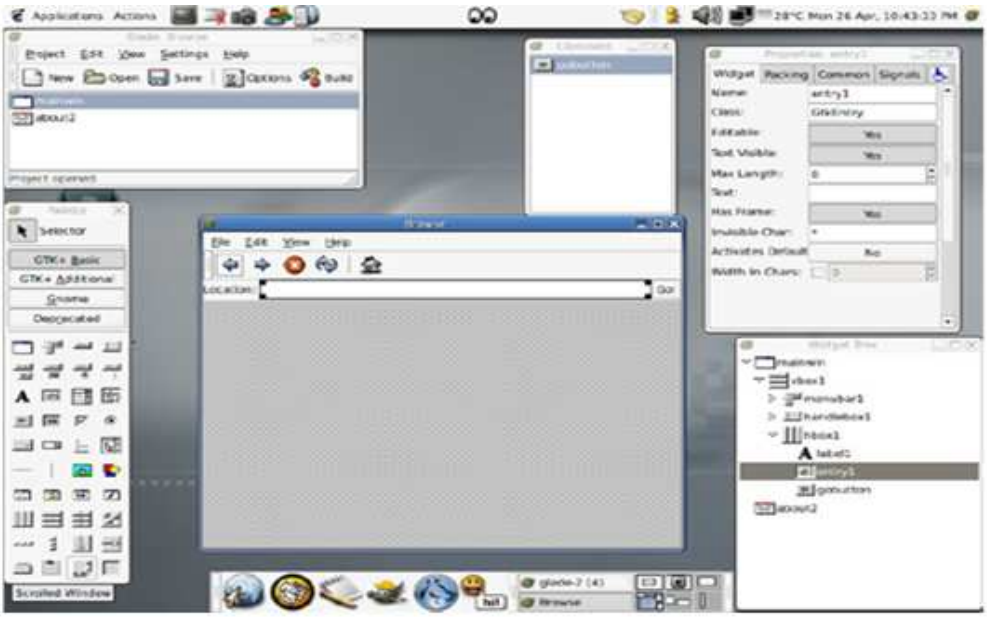


Figura 24. Glade corriendo en Linux

Pero el uso de Glade presenta, a su vez, un gran inconveniente pues genera código C y para utilizarlo debemos traducirlo a lenguaje ADA mediante la herramienta *gate*. Por tanto, cada modificación que hagamos en la interfaz, por pequeña que sea, deberá ser posteriormente traducida a ADA.

En este punto, se decide descartar el uso de ADA como lenguaje de programación. La razón principal consiste en que nos obliga a seguir una metodología muy compleja: se debe diseñar la interfaz de usuario de forma gráfica mediante Glade, traducir esa información a Ada mediante gate, una vez producidos esos ficheros programar la aplicación propiamente dicha, bien con un editor de textos bien con un IDE, y una vez programado compilar con el compilador GNAT

Al descartar el uso de ADA también se decide abandonar la idea de un simulador multi-robótico. La idea de un simulador con capacidad para representar y explotar escenarios multi-robóticos no es retomada hasta mucho después, cuando se consigue la creación de un simulador aunque sólo pudiera mover un único robot. De este modo, no es tan importante un lenguaje con facilidades orientadas hacia la multitarea. Por esta razón, se abre el abanico de posibilidades y se decide escoger entre lenguajes de propósito general orientados a objetos. Las opciones destacadas fueron: C++, C#, Object Pascal, Visual Basic, PHP, PowerBuilder y Python.

Teniendo en mente que el proyecto iba a requerir un lenguaje con suficiente potencia como para poder representar gráficos de escenarios en 3 dimensiones se descarta usar Visual Basic.

La mayoría de las aplicaciones que hacen uso de gráficos tridimensionales están escritas en C++ o en C#, por tanto, hubiera sido una buena opción escoger

un lenguaje como Visual C++. Sin embargo se decidió, finalmente, programar en Delphi. Las razones son sencillas, por un lado genera ejecutables suficientemente rápidos (aunque mínimamente más lento que Visual Basic), por otro lado porque Delphi usa Object Pascal, que no es ni más ni menos que una extensión orientada a objetos del lenguaje Pascal clásico. Las ventajas de utilizar Object Pascal son que la sintaxis es más clara, cuidada y fácil de leer que C y su extensión orientada a objetos ofrece potencia suficiente como para equipararse a los lenguajes orientados a objetos más recientes y modernos. Hay que añadir que el programador del código de la versión mono-robot de **RobotScene** estaba familiarizado con la programación de aplicaciones en Pascal, tanto MS-DOS con Turbo Pascal como en Windows con Delphi, lo que reduce el tiempo de desarrollo del proyecto.

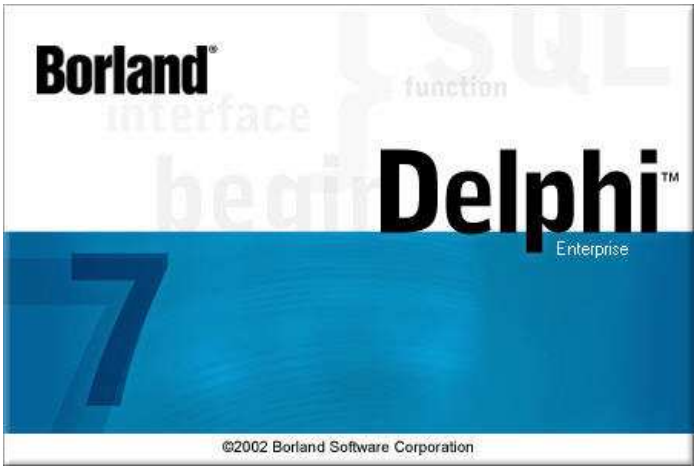


Figura 25. Borland Delphi 7

Una vez escogido el lenguaje de programación hay que pensar cómo se realizaran las operaciones gráficas requeridas para el funcionamiento del simulador. Teniendo en cuenta que la parte gráfica es de vital importancia en un proyecto de estas características, es el momento de decidir sobre los elementos necesarios para conformar dichos gráficos.

Con la idea de abstraer al programador del hardware de la máquina, lo que conlleva grandes ventajas como mejorar la facilidad para portar aplicaciones a otros tipos de hardware, ya que permiten realizar una implementación independientemente del tipo de tarjeta gráfica, o facilitar la programación con su consiguiente reducción de tiempo, se crearon las APIs (Application Programming Interface).

Es el momento de elegir entre las dos APIs más utilizadas: OpenGL o Direct3D, teniendo en cuenta, que estas dos APIs no son intercambiables.

OpenGL es una biblioteca gráfica desarrollada originalmente por Silicon Graphics Incorporated (SGI). OpenGL significa Open Graphics Library. Entre sus características destacan que es una API sencilla y compacta, mutiplataforma (Linux, Unix, Mac OS, Microsoft Windows, etc.), que se va actualizando mediante extensiones para aprovechar las crecientes evoluciones tecnológicas.

Por otra parte, Direct3D es un API producida por Microsoft que proporciona una funcionalidad similar a OpenGL aunque no se utiliza estrictamente en los mismos ámbitos como veremos más adelante. Forma parte de la familia de APIs DirectX, destinada sobre todo a la programación de videojuegos.

En cuanto a la portabilidad OpenGL supera ampliamente a Direct3D. Mientras que Direct3D es un sistema cerrado de Microsoft, OpenGL es abierto o libre, como su propio nombre indica, y por tanto, esta portado a numerosos sistemas operativos. Direct3D está sólo implementado en la familia de sistemas operativos de Microsoft Windows. Por su parte, OpenGL tiene implementaciones o ports en gran variedad de aplicaciones. Exceptuando Windows, el resto de sistemas que permiten aceleración gráfica mediante hardware han elegido a OpenGL como la API primaria.

En lo referido a la sencillez de uso, OpenGL vuelve a ganar a Direct3D. En los primeros pasos de Direct3D algunos programadores de videojuegos se quejaron de su dificultad de uso, como por ejemplo, John Carmack creador de Quake. Finalmente se actualizó Direct3D para mejorar la facilidad para programar. Las características de Direct3D derivan de las características que proporciona la tarjeta gráfica, mientras que por su parte, OpenGL está diseñado para ser un sistema de renderizado tridimensional que puede ser acelerado mediante hardware.

Los fabricantes de tarjetas gráficas diseñan sus productos de forma que sean compatibles con ambas APIs, sin embargo, cada una de las dos APIs se utilizan generalmente en distintos ámbitos. Por ejemplo, OpenGL ha sido siempre la API más utilizada en el mercado de las aplicaciones gráficas profesionales, entre otras cosas debido a que muchas características de OpenGL que no son útiles para la programación de videojuegos sí que lo son a la hora de realizar aplicaciones profesionales en el sector de gráficos tridimensionales, mientras que DirectX es ampliamente utilizada en la creación de videojuegos ya que ha sido diseñada para el acceso al bajo nivel del hardware y su explotación.

Por todas estás razones se decidió, en el desarrollo de la versión mono-robot, usar OpenGL, ya que RobotScene no iba a ser un videojuego sino un simulador, es decir, una aplicación que utiliza gráficos en 3D. Otra de las razones importantes por las que se decidió escoger OpenGL fue que si la API escogida hubiera sido DirectX no se podría realizar jamás un port del simulador robótico a Linux.

Figura 26. Logotipos de OpenGL y DirectX.

Como se ha comentado anteriormente la API gráfica permite al programador abstraerse del hardware y controlar el acceso a dicho hardware mediante llamadas a los drivers de video. Sin embargo programar una aplicación gráfica directamente sobre la API puede ser una tarea bastante compleja, es por ello por lo que se utilizan motores gráficos, que añaden una capa de software que está por encima del API y que proporciona una estructura orientada a objetos al programador final de la aplicación, liberando a éste del trabajo directo con el API de video. El motor gráfico gestiona la representación del escenario, de sus componentes y de los recursos de video auxiliares para llevar a cabo dicha tarea, (sprites, texturas, buffers de video, vertex buffers, etc.). El motor gráfico es el responsable de solucionar como y cuando se pinta algo en pantalla y de activar los diferentes estados del dispositivo de vídeo.

En este momento de desarrollo del proyecto se decidió cuál iba a ser el motor gráfico que sustentaría la representación gráfica de nuestra aplicación. Por todo lo dicho anteriormente, el motor debe estar realizado sobre el API OpenGL y ser compatible con Delphi. Hay motores gráficos de diferentes tipos, unos gratuitos y otros con licencias a precios poco asequibles, asi que, debido al ámbito de la aplicación, lo lógico era escoger un motor gratuito y libre. De las diferentes opciones existentes a la hora de decidirse por un motor gráfico con las características mencionadas anteriormente destacan GLXStreem, GLScene, DelphiX y Asphire.

Asphire quedó inmediatamente descartado debido principalmente por dos razones. La primera es que es un motor diseñado para gráficos en 2 dimensiones, la segunda es que esta desarrollado bajo DirectX. El siguiente motor en quedar descartado fue DelphiX, pues como su propio nombre indica utiliza DirectX y adicionalmente esta pensado para el desarrollo de videojuegos. GLXStreem está basado en OpenGL, lo que podría ser una buena opción si no fuera por que está altamente enfocado a la programación de videojuegos.

GLScene es un conjunto de librerías basadas en OpenGL, de ámbito mucho más general que otros motores gráficos como GLXStreem. La estructuración de clases es mucho más avanzada que el resto de motores lo que facilita enormemente la tarea del programador. GLScene no se considera un motor como tal, sino una librería de clases con las que se puede desarrollar rápidamente un

motor propio, la estructuración que se realiza de la escena y su jerarquía es muy superior a la de otros motores.

GLScene no está orientado a la programación de videojuegos sino a la programación de aplicaciones, como ejemplos podemos recordar que GLScene ha sido utilizado en aplicaciones médicas, como visores volumétricos de imágenes médicas, programas CAD para el diseño de cocinas, cuartos de baño, más otras aplicaciones como un simulador de soldadura, programas para el estudio de la litología e incluso simuladores de brazos robóticos. También ha sido utilizado para la programación de videojuegos, reconociéndose mundialmente como la mejor librería para gráficos tridimensionales en Delphi.

GLScene es un proyecto publicado bajo licencia MPL (Mozilla Public license), licencia de código abierto y software libre, que a éstos efectos es una licencia equivalente a la GPL (Gnu Public License) pero con código abierto.

Por todo lo expuesto se consideró más que recomendable el uso de GLScene para el desarrollo del simulador robótico. El uso de GLScene dio la ventaja de poderse centrar en aspectos puramente robóticos sin tener que conocer los entresijos de OpenGL, lo cual inevitablemente hubiera consumido una gran parte del tiempo destinado a la realización del proyecto.

Figura 27. Logo GLScene

Una vez escogidos el lenguaje de programación y las herramientas para generar gráficos en 3 dimensiones se constató que iba a ser necesaria la utilización de scripts para dos usos principales. Un script es una herramienta que permite ejecutar código externo en una aplicación sin la necesidad de volver a recompilar toda la aplicación ni de tener que distribuir el código fuente.

Por un lado, se decidió que fuera el usuario el que implementaraá el modelo geométrico inverso para calcular el valor de cada una de las articulaciones del robot para que éste pueda alcanzar una posición y orientación concreta. Si utilizamos un motor de script para este propósito, el usuario puede crear tantos robots como quiera si implementa el modelo inverso desde un script. Si no se utilizara un script la complejidad aumentaría bastante porque los modelos geométricos inversos deberían ser programados sobre el código fuente del simulador, y por tanto, habría que recompilar toda la aplicación cada vez que se añadieran nuevos modelos inversos para nuevos robots.

Por otro lado, si el programa cuenta con un motor de scripts, será muy fácil poder dotar al programador de un lenguaje textual para programar las tareas robóticas. En este caso es el script el que ejecuta funciones propias del programa.

Las alternativas que se ofrecen para incluir un motor de scripts en Delphi son DWS, PaxScript y PascalScrip. Otra última alternativa podría ser la utilización de lenguajes de script como Python, sin embargo, esta opción queda descartada inmediatamente porque obligaría al usuario del simulador a instalarse Python en su máquina, y por tanto, nuestra aplicación no sería autocontenida.

PaxScript es un motor de lenguaje de script interpretado. Como ventaja principal destaca ser un intérprete de cuatro lenguajes, Basic, C, Pascal y JavaScript, por lo que aunque nuestra aplicación esté escrita en Delphi el usuario podría especificar el lenguaje en el que quiere programar y elegir entre estas cuatro opciones. Presenta funcionalidades para la programación orientada a objetos. Pero el gran inconveniente que presenta PaxScript consiste en que no es libre, por lo que hay que adquirir licencia para su utilización. Su uso es mucho más complejo que las otras dos opciones, y no cuenta con una comunidad tan activa como los otros dos casos, por lo que quedó descartado su uso.

DWS (Delphi Web Script) es un motor de scripts capaz de interpretar código de una sintaxis muy similar a Pascal. DWS presenta algunas limitaciones como el uso de punteros, la sobrecarga de funciones o la definición de clases mediante el script. Debido a que es código abierto está sufriendo continuas mejoras y añadidos constantemente. A pesar de ser interpretado, es un motor bastante rápido.

PascalScript de RemObjects Software, es un motor de scripts gratuito, que se distribuye con el código fuente, capaz de interpretar código con sintaxis de Pascal. Como punto a favor para decidirse por este motor, cabe destacar que es el motor más rápido de los tres mostrados, y además tiene la ventaja de que los scripts pueden ser interpretados o 'compilados'. Aunque la compilación de scripts no produce código ejecutable, dicha compilación se traduce en la creación de una serie de datos internos que utiliza el motor de script de forma que sólo se necesite acceso al código fuente del script en momento de la compilación, resultando así más rápida la ejecución que si el script fuera interpretado. Tiene la desventaja de no contar con ningún tipo de documentación, aunque esto no debe ser necesariamente un problema puesto que en su distribución se incluyen numerosos ejemplos en los que se puede aprender el funcionamiento. Adicionalmente existen wikis y páginas donde la comunidad resuelve problemas relativos al uso de dicho motor de scripts. A su vez RemObjects Software pone a disposición de sus usuarios un contacto para resolver dudas por correo electrónico.

Por todo lo expuesto anteriormente, se decidió que el motor de scripts que se iba a utilizar fuera Pascal Script. En la aplicación será necesario resolver de forma constante y continuada el modelo inverso del robot, así que se requiere de cierta velocidad a la hora de ejecutar los scripts. Pascal Script es más rápido que los otros dos y además cuenta con la ventaja de que se puede compilar, aumentando por tanto su velocidad (respecto a lenguajes interpretados) debido a que es más rápido un lenguaje compilado que uno interpretado.

Fig 28: RemObjects Pascal Script

Como conclusiones finales de este apartado se resumen las soluciones finales escogidas en el desarrollo de la versión mono-robot de **RobotScene**:

- Se utilizó Delphi como lenguaje de programación para crear el código fuente de la aplicación.
- Para la creación de gráficos se utilizaron las librerías GLScene que están basadas en la API OpenGL
- Para que el usuario final de la aplicación pudiera programar el modelo inverso de su robot y las instrucciones y tareas que dicho robot debe ejecutar en un escenario robótico se utilizó el motor de scripts Pascal Script de RemObjects Software.

En la nueva versión multirobot de **RobotScene** se ha intentado en todo lo posible mantener estas decisiones por varios motivos. En primer lugar, las decisiones adoptadas siguen pareciendo a día de hoy las más correctas. En segundo lugar, estas decisiones afectan a partes importantes del programa, y por tanto, su cambio repercutiría en varias y numerosas partes del código fuente, con el consiguiente esfuerzo de remodelar gran parte de la aplicación y el tiempo que se necesita para ello.

4 REQUISITOS DE DISEÑO

El objetivo de la modificación de la versión mono-robot de RobotScene es, como se ha señalado más arriba, la consecución de un escenario multi-robótico que simule un ambiente industrial en el que varios robots realicen simultáneamente las tareas que se les ordenen y se comuniquen entre ellos.

En la versión que se va a modificar es posible añadir uno o varios robots al escenario, pero sólo uno de ellos – el último robot añadido – permanece activo. Este robot usa de forma excluyente el único script de programa disponible y la gestión de E/S.

Es posible seleccionar en **RobotScene** mono-robot uno u otro de los sucesivos robots añadidos mediante la correspondiente herramienta de selección pero, una vez lanzada la ejecución del programa, no es posible cambiar de robot activo mientras el programa se ejecuta

Por tanto, diseñar un escenario multi-robot requerirá, en lo que se refiere a los scripts de programación

- Disponer de un *script* por cada robot
- Que el *script* se abriese al seleccionar el robot correspondiente previa activación de la herramienta de edición de programa
- Que el lanzamiento de la ejecución de cada aplicación se realice desde su *script* correspondiente (botón "*play*")

Los requisitos anteriores sugieren que su implementación deberá lograrse mediante la creación y gestión de una ventana de programación por cada robot. Si se establece un número máximo de tres robots, se dispondrá de tres ventanas de programación que, en caso necesario, estarán abiertas de forma simultánea si se ha pulsado sobre el correspondiente robot.

La gestión de las ventanas será independiente: el programa será lanzado desde el botón "play" que le corresponda, afectando cada programa a su robot asociado.

En un contexto multirobot, la coordinación entre los diferentes robots es clave para conseguir la emulación de escenarios realistas. Las opciones de coordinación más destacables son la sincronización mediante semáforos y la sincronización mediante la emulación de cableados de E/S binarias.

La segunda opción supone una emulación mas realista de la realidad industrial: en ese entorno, el cableado de sincronización es la forma de sincronizar elementos con softwares potencialmente muy diferentes.

Además, ya se cuenta instrucciones de E/S binarias; para implementar la sincronización se necesitará únicamente diseñar un sistema que permita definir y mantener el cableado entre los módulos de E/S de cada robot.

Por las dos razones expuestas, en la versión multi-robot de RobotScene se optará por la sincronización por emulación de cableado de E/S binarias implementando funciones de lectura/escritura a nivel de lenguaje robot, así como un sistema de definición y mantenimiento del mencionado cableado. De esta forma, cada conexión establecida sería equivalente a un semáforo sobre el que únicamente pueden actuar - de manera limitada o unidireccionalmente - los robots (hilos) involucrados

Con tal fin, deben crearse o modificarse 2 interfaces (o formularios):

1. Formulario de edición/monitorización del cableado, con todas las herramientas necesarias para la edición gráfica y almacenamiento del cableado.
2. Un formulario específico para la monitorización gráfica de las E/S y explotación manual de las entradas.

Ambos formularios o interfaces deben ser sencillos de manejo y permitir realizar la sincronización de robots de forma efectiva; por ello deben reunir los siguientes requisitos funcionales

<u>Formulario de cableado</u>

- Pantalla con las herramientas básicas de archivo (nuevo, abrir cableado existente, guardar cableado)
- La pantalla aparecerá al pulsar un botón específico de la barra de herramientas (con un icono que evoque el cableado entre 2 borneros dispuestos en vertical), independientemente de que haya abierto algún programa de robot.
- En la pantalla deberán aparecer tantos módulos de E/S como robots haya en el escenario.
- El cableado se llevará a cabo mediante el ratón, al pulsar-arrastrar-soltar desde/hacia los borneros de E/S correspondientes.

<u>Formulario de monitorización de E/S</u>

- La pantalla aparecerá al pulsar la herramienta "E/S", aunque esté abierto algún programa de robot.
- En la pantalla deberán aparecer tantos módulos de E/S como robots haya en el escenario.
- Cada módulo de E/S dispondrá de 2 zonas claramente individualizadas (separadas): una para entradas y una para salidas.
- Cada módulo de entradas dispondrá de 2 regletas: interruptores para la actuación manual por parte del operario y leds indicadores del estado de la entrada.
- La prelación entre el bornero y la regleta de interruptores es tal que si se usa un borne con fines de cableado, el interruptor correspondiente deja de estar habilitado.

- Cada módulo de salidas dispondrá de 2 regletas: bornero de conexionado y leds indicadores del estado de la salida.

5 ANÁLISIS DE SOLUCIONES

El contenido de este apartado se refiere al estudio de las diferentes soluciones técnicas que se pueden emplear en el diseño de la versión mutil-robot de **Robotscene**. Llegados a este punto conviene recordar que, debido a que se parte de una versión anterior del programa, en algunas ocasiones nos hemos visto empujados a asumir o continuar con decisiones tomadas previamente.

La primera idea fundamental a la hora de rediseñar el programa ha sido intentar mantener las alternativas escogidas en su momento, primero porque a priori las decisiones tomadas fueron las correctas, y en segundo lugar porque hay una serie de elecciones importantes como son el lenguaje de programación o el motor de gráficos que, de ser modificados, inducirían cambios en gran parte del código del programa y, por tanto obligaría a rehacer gran parte de la aplicación hasta tal punto que podría ser más sencillo crear desde cero una aplicación similar. Resumiendo, trataremos de alcanzar el objetivo final utilizando GLScene y PascalScript.

En este apartado se analizarán las estrategias elegidas para poder llevar a cabo las tres tareas más importantes para alcanzar los objetivos previamente descritos. Estas tres tareas principales son la creación y gestión de una ventana de programación por cada robot, la ejecución por parte de los robots de instrucciones de programa de forma simultánea y la consecución de una comunicación efectiva entre robots que simule un cableado real entre robots.

5.1 CREACIÓN Y GESTIÓN DE LAS VENTANAS DE PROGRAMACIÓN.

En la anterior versión de RobotScene se utilizaba una ventana de programación para indicar al robot mediante líneas de código o texto qué acciones o movimientos debía acometer para la ejecución de una determinada tarea. Inicialmente, el simulador robótico estaba desarrollado para la explotación de un único robot; por ello, como es lógico, el simulador poseía una única ventana de programación. La ventana de programación se asociaba automáticamente con el último robot seleccionado. De este modo, el programa podía explotar tantos robots como existieran en el escenario, pero de manera individual y no simultánea.

Para una versión multi-robot de dicho programa se decidió que lo ideal sería la creación y gestión de una ventana de programación por cada robot. De esta forma sería posible asociar cada ventana con un único robot. Dicha ventana de programación se crearía o destruiría automáticamente con la creación o destrucción de un robot.

Para conseguir este objetivo se decide que lo mejor y más lógico sería copiar repetidas veces la ventana de programación. Al proceso de copiar se le denomina en informática *instanciar*. En un lenguaje de programación orientado a objetos *instanciar* es crear un objeto perteneciente a una clase (se dice que *se instancia la clase*).

El objeto creado posee los mismos atributos, características, propiedades, y métodos de la clase a la que pertenece. Es decir, si instanciamos la ventana de programación, por cada robot que se cree conseguiremos tantas ventanas de programación como robots existan. Además, cada ventana de programación será independiente de las otras.

Una de las ventajas que se consiguen al instanciar las clases es que se pueden tener tantos objetos de dicha clase como se deseen sin que ello suponga casi esfuerzo al programador. Sin embargo, el programador debe ser consciente de que cada copia o instancia de una clase consume una cierta cantidad de memoria y que, por tanto, su esfuerzo debe ir enfocado a la creación correcta de cada objeto, y lo que es más importante, a su destrucción, de manera que se libere la memoria que se ha utilizado para albergar la copia.

Una vez establecido que se copiarían las ventanas realizando instancias de su clase, se decide que cada instancia de ventana de programación se guardará en un vector de datos. Para la correcta asociación de cada ventana con su robot correspondiente se creará también un vector para almacenar las variables relativas a cada robot. Como se ha comentado, cada instancia de ventana ocupa cierta cantidad de memoria; en este punto se decide que el número máximo de robots en el escenario sea de tres.

Realizado todo lo anterior, se supuso, erróneamente, que cada instancia de la ventana podría ejecutar un programa robótico, debido a que cada clase de la ventana de programación contenía a su vez una clase del motor de scripts. Sin embargo, no se percibió, inicialmente, que cada motor de scripts iba a requerir de un hilo de ejecución para su correcto funcionamiento.

En el siguiente apartado analizamos las distintas soluciones consideradas para conseguir la ejecución simultánea de varios programas robóticos.

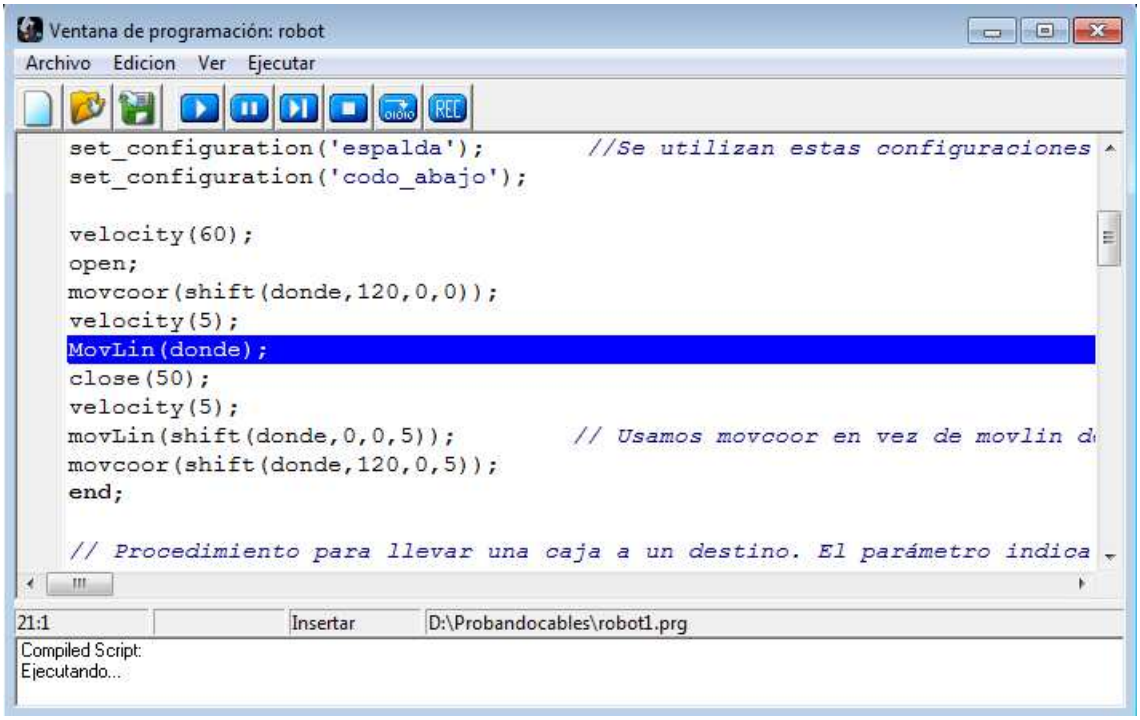


Figura 28. Captura de imagen de la ventana de programación

5.2 CONSECUCIÓN DE LA CONCURRENCIA

Se inicia este apartado definiendo lo que es la concurrencia informática para tener más claro por qué es necesaria y para qué quiere conseguirse. Llamamos computación concurrente a la simultaneidad en la ejecución de múltiples tareas. Estas tareas pueden ser un conjunto de procesos o hilos de ejecución creados por un único programa.

A menudo, durante el proceso de desarrollo de soluciones para conseguir la concurrencia, se ha utilizado una estrategia de ensayo y error que, de forma secundaria, ha permitido ir conociendo cada vez más acerca de la programación concurrente o multitarea. Por otra parte, cuando se inició el desarrollo de la versión multi-robot se supuso que el programador no necesitaría programar la implementación de la concurrencia, sino que sería el propio programa el que permitiera la ejecución paralela de diferentes eventos.

La primera solución que quedó descartada fue la instanciación de las ventanas de programación. En nuestro programa iba a ser necesaria tal instanciación - como se ha argumentado en el apartado anterior -, pero sin embargo, no se conseguía ejecutar varios programas robóticos simultáneamente instanciando únicamente las ventanas. En este punto se comprueba que cuando un script se esta ejecutando, dicho script toma posesión de la aplicación, impidiendo que se pueda ejecutar al mismo tiempo un segundo script.

Por tanto, han de buscarse nuevas alternativas. Entre ellas destacan dos: el uso de componentes en Delphi que eviten al programador escribir el código relativo a diferentes hilos de ejecución, o encontrar un motor de scripts que esté desarrollado específicamente para ejecutar varios scripts a la vez. Sin embargo, las dos vuelven a quedar descartadas por varios motivos.

Un componente en Delphi que abstraiga al programador de escribir código relativo a diferentes hilos de ejecución, o bien es difícil de encontrar, o si se encuentra, posiblemente no realice las acciones que el programador necesite para una tarea en concreto. Normalmente estos componentes están muy limitados, por ejemplo, un tipo de componente que se encontró permitía ejecutar hilos de forma simultánea y con cierta facilidad, pero presentaba como inconveniente que todos los hilos de ejecución creados debían ejecutar el mismo código y todos los hilos debían iniciarse o detenerse al mismo tiempo. Por tanto, carecía de sentido su utilización en nuestra aplicación.

Por otro lado, encontrar un motor de scripts que contemple expresamente la ejecución en multi-hilo constituye una ardua tarea que, probablemente, puede llegar a ser imposible de conseguir, puesto que se supone que es labor del programador de la aplicación realizar los oportunos cambios en su programa para ejecutar hilos de forma simultánea. Adicionalmente, migrar a un motor de scripts diferente incrementaba ampliamente el trabajo a realizar.

El cambio de motor de scripts obligaría a rehacer gran parte de la aplicación. Así, deberían efectuarse cambios en los controles de la ventana de programación, habría que familiarizarse con el nuevo motor de scripts y crear convenientemente

instrucciones para compilar y ejecutar scripts, deberían realizarse cambios en la parte de código relativa a la asociación del script con el lenguaje robótico. Después de efectuar dichos cambios, habría que probar si las nuevas librerías del motor de scripts son capaces de ejecutarse de forma concurrente, y seguramente, estas nuevas librerías no estarían diseñadas para tal fin, con lo que todo el trabajo realizado, habría sido en vano.

Estudiando nuevamente el problema, se decide que la mejor y única elección posible es mantener el motor de scripts actual (Pascal Script) y encargarnos personalmente de la ejecución simultánea de varios hilos. Para estos propósitos Delphi incorpora una clase llamada TThread que permite mediante instrucciones relativamente simples la creación de uno o varios hilos que se ejecutan de forma paralela con el hilo principal.

Delphi encapsula en la clase TThread la funcionalidad de los threads del API de Windows para poder manejarlos de forma sencilla. De esta manera crear un hilo se hace de forma relativamente fácil, tan sólo es necesario sobreescribir en un descendiente de la clase TThread el método Execute, que es definido como abstracto en el interfaz de la clase. Más tarde será necesario sincronizar los diferentes hilos de ejecución con el hilo principal para evitar así que varios hilos accedan a sitios de memoria de forma simultánea, teniendo en cuenta, además, que el VCL no soporta el acceso concurrente de múltiples hilos.

Sobre esta idea, el trabajo a realizar sería el relativo al encapsulado de un motor de scripts dentro de la clase TThread, previamente modificada a nuestro gusto para conseguir el correcto funcionamiento con nuestro simulador y después sincronizar hilos y gráficos con el hilo principal.

5.3 COMUNICACIÓN ENTRE ROBOTS

Llegados a este punto, la versión multirobot del programa ya es capaz de ejecutar varios programas robóticos de forma simultánea. Se crea convenientemente una ventana de programación por cada robot, que ejecuta, únicamente sobre el robot asociado, un programa robótico independientemente del resto de robots del escenario.

No se podría hablar con rigor de simulador con capacidad multi-robot si estos robots actuaran de forma independiente en el escenario. Si se observara un escenario con dos robots en movimiento pero sin comunicación directa entre ellos, no hablaríamos de una escena multi-robótica sino de dos escenas mono-robot. Por tanto es completamente necesario para el objetivo final del proyecto dotar a los robots de nuestro simulador de plena capacidad de comunicación entre ellos.

El trabajo en esta tarea consiste en conseguir que los robots puedan *hablar* o comunicarse entre sí de forma que se simule un cableado entre robots de un entorno robótico industrial real. Si se consigue emular un cableado real estamos dotando a nuestros robots de un sistema de comunicaciones que les permitirá interactuar entre ellos para la realización de trabajos coordinados entre sí.

De este modo pueden realizar tareas robóticas en las que intervengan varios robots, bien porque la tarea necesita de ambos robots para su consecución, o por ejemplo porque ambos robots ejecutan movimientos en el mismo espacio de trabajo, donde la comunicación es necesaria para evitar colisiones entre ambos robots.

Para conseguir este objetivo se presentan dos opciones principales: asociar entradas y salidas de cada robot mediante cuadros o tablas, y asociar dichas entradas y salidas de forma gráfica.

Debido a que nuestra aplicación ya cuenta con un motor de gráficos basado en OpenGL, se ha considerado que la mejor opción sería utilizar el propio GLScene para crear un motor de gráficos en dos dimensiones personalizado. Al realizar dicha unión de entradas y salidas entre diferentes robots mediante una interfaz gráfica se facilita al usuario la comprensión del esquema de cableado. El usuario sólo necesitara arrastrar cables desde una salida de un robot a la entrada de otro. El resultado es estéticamente agradable y el cableado se realiza de una manera mucho más intuitiva. Además la creación de un motor gráfico en dos dimensiones a partir de GLScene resulta relativamente sencillo.

En la versión mono-robot de **RobotScene** ya se contaba con un cuadro de entradas y salidas que permitía al robot activar o desactivar sus salidas. De esta manera, el robot podía dar información relativa a su estado activando o desactivando sus salidas y podía recibir información a través de sus entradas de forma que podría quedarse en un estado de letargo o espera en función de lo que decidiera el programador de la tarea robótica. En esta versión multi-robot se ha modificado dicha ventana de entradas y salidas para ajustarlo a la nueva condición multi-robot.

Para la comunicación de esta ventana de entradas y salidas con la interfaz gráfica que define el cableado se decidió la inclusión de una matriz de cableado por cada robot. Esta matriz guardará la información relativa de las entradas de dicho robot con las salidas de los restantes robots del escenario. Estas matrices se guardarán dentro de un vector que almacenará la información relativa al cableado. Para guardar toda esta información referida al cableado una vez que el programa se cierra se ha implementado un sistema de ficheros.

De esta forma se ha conseguido emular un cableado real de forma visual y muy intuitiva para el usuario.

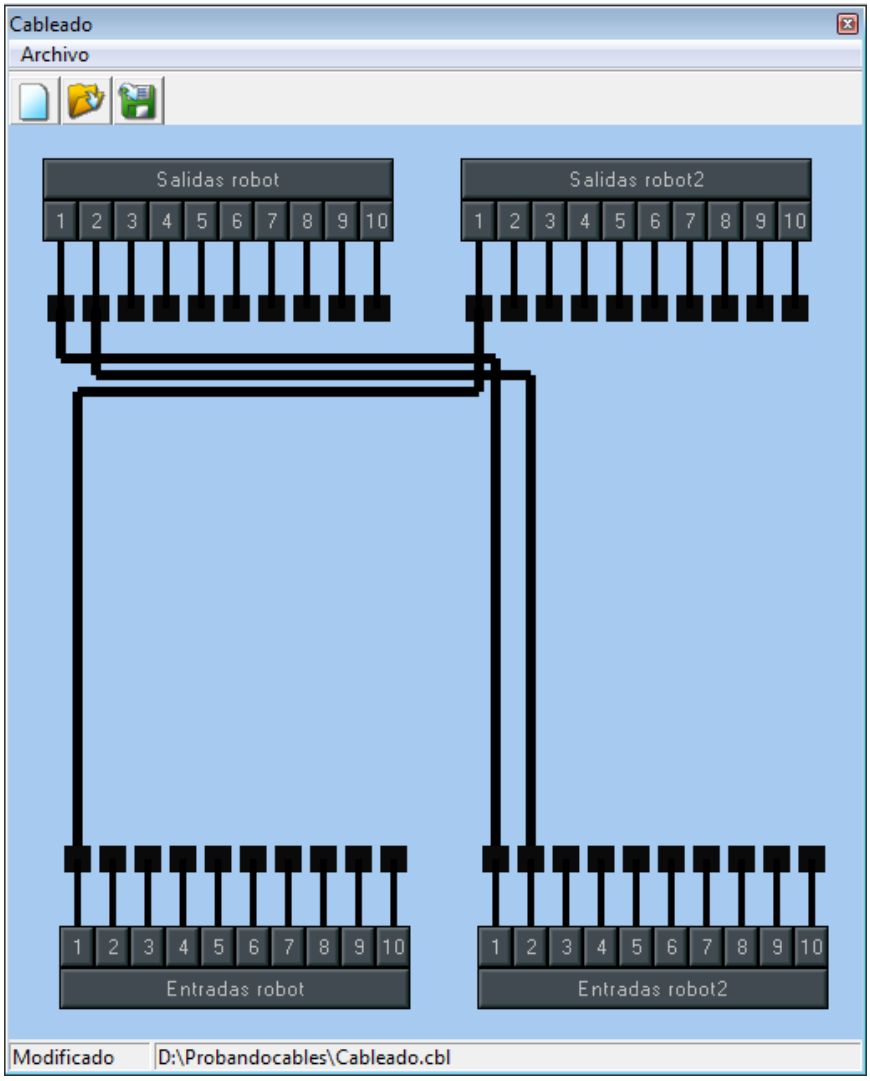


Figura 29. Ventana cableado RobotScene

6 DESCRIPCIÓN DE LA APLICACIÓN

Para dotar a la versión mono-robot de **RobotScene** de la capacidad de simular escenarios multi-robóticos, sólo ha sido actualizado el último de los tres módulos. Por ello, la versión multi-robot sólo difiere de la original en el constructor de escenarios que es el módulo que ha sido modificado y actualizado. Los apartados siguientes de la Memoria hacen referencia únicamente a las modificaciones introducidas en ese módulo

El constructor de escenarios es el más importante de los tres módulos que componen el simulador. Como su propio nombre indica, permite la construcción de escenarios robóticos añadiendo a la escena robots y objetos creados previamente con los otros dos módulos. Adicionalmente este último módulo permite, como es lógico, no sólo la construcción del escenario sino también su explotación posterior. De esta forma, podemos dotar a los robots de movimientos mediante guiado, registro de puntos para crear una secuencia o mediante programas robóticos textuales.

En este apartado se describirá el constructor de escenarios, haciendo hincapié en sus características y funcionalidades, y repasando su código fuente. Una vez acabado este apartado el lector tendrá una idea general sobre lo que es **RobotScene** (versión multi-robot), concretamente del último y más importante de sus módulos, y sobre las capacidades técnicas y de simulación que ofrece.

6.1 DESCRIPCIÓN DE LA INTERFAZ DE USUARIO

En esta descripción de la interfaz, se describirán las posibilidades del constructor de escenarios de manera que un usuario no familiarizado con el programa pueda entender, a grandes rasgos, para qué sirve y cómo funciona, así como las opciones que ofrecen los menús y barras de herramientas.

La imagen siguiente muestra la interfaz general de usuario:

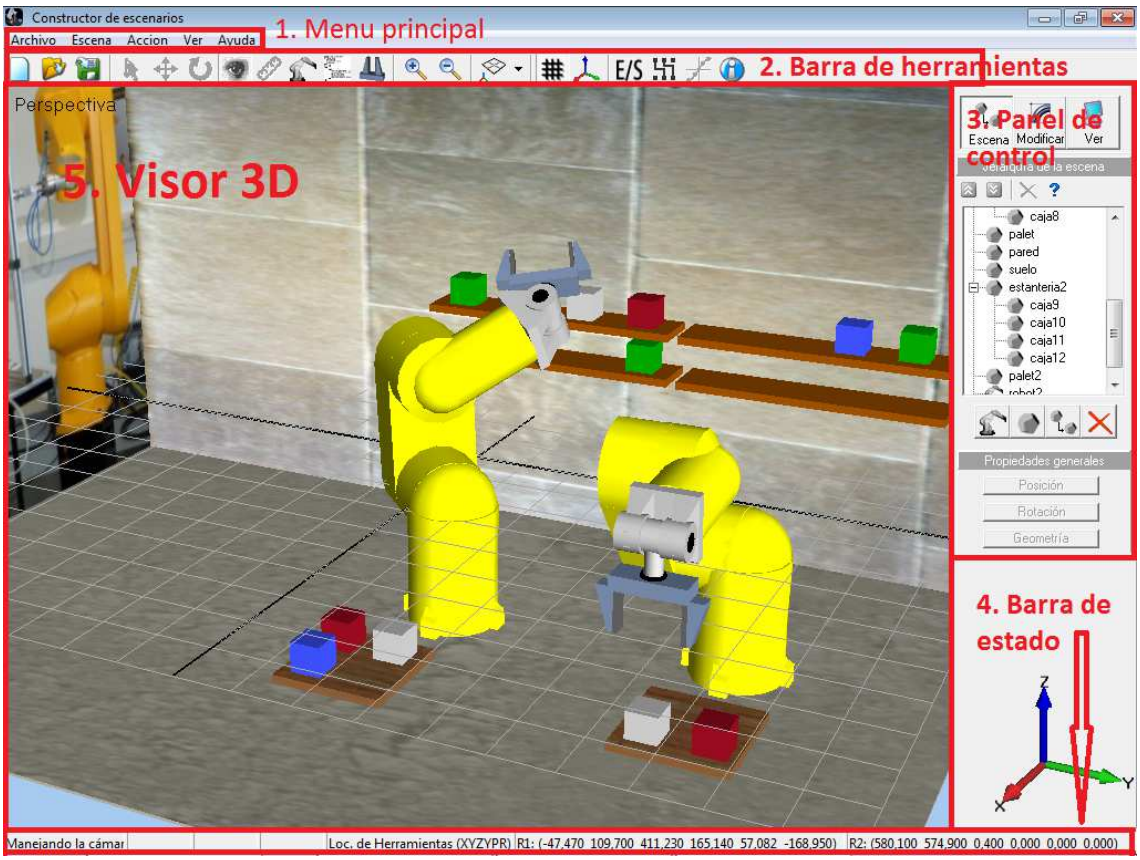


Figura 30. Captura de la interfaz de usuario

Tal como se puede observar en la imagen, el módulo constructor de escenarios presenta las siguientes zonas:

1. **<u>Menú principal.</u>**

Desde el menú principal podemos acceder a prácticamente todas las funcionalidades del programa. Está compuesto por los siguientes elementos:

Archivo: En esta opción del menú el usuario puede cargar un archivo de escenario (*.esc*), guardar el escenario, crear un nuevo escenario o salir del programa.

Escena: Mediante esta parte del menú el usuario tiene la opción de añadir elementos al escenario, el usuario puede añadir un nuevo sólido, añadir un robot, o eliminar el objeto seleccionado.

Acción: Este menú permite al usuario seleccionar los diferentes modos de operación. Por un lado nos encontramos las acciones relativas a los objetos del escenario, teniendo la posibilidad de seleccionar objetos, trasladar objetos y rotar objetos. Por otro lado encontramos acciones relativas a gráficos como son manejar la cámara y medir distancias en el escenario. Finalmente, el usuario puede realizar acciones relativas a los robots. Dichas acciones son *guiar robot*, *programar robot* y *añadir una herramienta al robot*.

Ver: Desde este menú se acceden a diferentes opciones referentes a la visualización de la escena. Las dos primeras opciones *acercar* y *alejar* permiten ajustar la distancia de la cámara con el escenario actuando como zoom. También se encuentran en este menú diferentes opciones gráficas relativas al modo de representación de los objetos: *suavizado*, *sin suavizar* y *alambres*. *Suavizado* permite ver los objetos con un buen acabado gráfico mientras que *alambres* deja en segundo plano las opciones gráficas para mejorar el rendimiento. Mediante el menú *ver* se posibilita la opción de cambiar de vista pudiendo elegir entre cuatro vistas diferentes: perspectiva, cenital (XY), lateral (YZ), frontal (XZ). Las últimas opciones de este menú permiten abrir las diferentes ventanas que componen el constructor de escenarios: ventana de registro, ventana de entradas y salidas (E/S) y la ventana de cableado.

Ayuda: Menú informativo que presenta las siguientes opciones: *acerca de* permite al usuario informarse sobre los créditos de los creadores del programa, *configuración* permite al usuario cambiar ciertas opciones gráficas como el antialiasing, el color de fondo o la profundidad de color.

2. Barra de herramientas

Como suele ser habitual en los programas informáticos, el constructor de escenarios cuenta con una poderosa barra de herramientas, que a disposición del usuario, permite realizar todas las acciones del constructor de escenarios.

Figura 31. Barra de herramientas del constructor de escenarios

A continuación se detalla individualmente la acción que realiza cada uno de los botones.

 Crea un nuevo escenario.

 Carga un escenario previamente guardado.

 Guarda el escenario actual.

 Selecciona objetos del escenario.

 Traslada objetos en el escenario.

 Rotar objetos del escenario.

 Permite al usuario manejar la cámara libremente en el escenario

 Mide distancias en el escenario.

 Guia un robot del escenario.

 Abre la ventana de programación al pinchar sobre el robot.

 Permite añadir o eliminar una herramienta a un robot.

 Acerca la cámara (zoom in).

 Aleja la cámara (zoom out).

 Selecciona las diferentes vistas de la cámara: perspectiva, cenital (XY),lateral (YZ), frontal (XZ).

 Activa u oculta la rejilla en el escenario.

 Permite observar los ejes de posición y orientación de los objetos del escenario.

 Permite observar la ventana de entradas y salidas de los robots.

 Activa la ventana de cableado.

 Activa la ventana de registro por puntos cuando se esta guiando un robot.

 Permite observar información relativa al driver OpenGL y tarjeta gráfica.

3. Panel de control

El constructor de escenarios de **RobotScene** cuenta con un panel de control situado en la parte derecha de la ventana. Este panel de control sirve principalmente para controlar la creación de escenario.

El panel de control se compone realmente de tres subpaneles. Para cambiar entre subpaneles se han habilitado tres botones que se muestran en la siguiente imagen:

Figura 32. Botones *Escena*, *Modificar* y *Ver*, creados para alternar entre subpaneles.

A continuación se detalla la forma y contenido de cada uno de los subpaneles que conforman el panel de control.

Panel Escena

Es el más importante de los tres, por sus funcionalidades; es también el más utilizado. Su función principal es la de creación del escenario. Está divido en dos partes.

Figura 33. Imagen del panel *Escena*.

La primera de estas dos partes está creada bajo el título de *Jerarquía de la escena*, como indica su nombre, permite establecer el contenido del escenario de forma jerarquizada, estableciendo una preferencia de los elementos que componen el escenario de forma arbórea.

Más adelante se hablará con más detalle acerca de la jerarquía de la escena. Por el momento, es suficiente conocer para qué sirve cada uno de los botones del panel.

 Permite subir de posición un elemento del árbol dentro de su jerarquía.

 Permite bajar de posición un elemento del árbol dentro de su jerarquía.

 Elimina un elemento del árbol de jerarquía, y por tanto, del escenario.

 Botón ayuda.

 Añade un robot a la escena.

 Añade un objeto a la escena.

 Añade un hijo a un objeto de la escena.

 Borra completamente el escenario.

La segunda de las partes se denomina *Propiedades generales* y permite ajustar numéricamente la posición y la orientación de los objetos del escenario, así como cambiar su geometría.

Panel Modificar

El panel *Modificar* nos permite activar efectos gráficos como sombras o cielo y permite activar o desactivar la detección de colisiones.

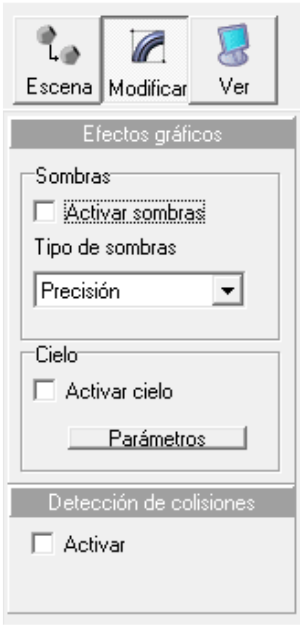


Figura 34. Panel *Modificar*.

Panel Ver

El panel *Ver* nos permite activar efectos gráficos relativos a la cámara. La luz puede activarse y desactivarse, así como, cambiar su posición y su color. Otra de las opciones de este panel es cambiar la vista de la cámara.

Figura 35. Imagen del panel *Ver*.

Adicionalmente el botón que se muestra a continuación permite capturar la imagen del visor de perspectiva en tres dimensiones para posteriormente guardarlas en el disco duro:

Figura 36. Botón que permite capturar la imagen del visor 3D.

4. Barra de estado

La barra de estado del constructor de escenarios muestra en su parte lateral izquierda información relativa al modo de operación actual del escenario.

En la zona central muestra la información relativa al objeto seleccionado, es decir, su orientación y posición. Por último en la parte lateral derecha hay un gran espacio reservado para mostrar la orientación y posición de la herramienta del robot.

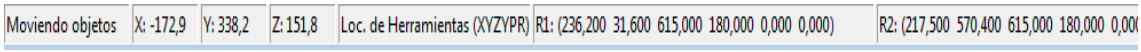


Figura 37. Barra de estado del constructor de escenarios.

5. Visor 3D

El visor 3D ocupa gran parte de la ventana principal del constructor de escenarios. Dicho visor es la pieza clave del constructor de escenarios porque nos muestra la representación visual en 3 dimensiones del escenario, en otras palabras, es donde se observará el movimiento de los robots de la aplicación. El usuario puede interactuar con dicho visor para realizar ciertas operaciones con los objetos del escenario, como podría ser moverlos o rotarlos. Aparte del escenario propiamente dicho se pueden ver otros elementos como ejes o rejillas que facilitan la compresión de la escena.

El visor 3D es utilizado por tanto, como elemento para la visualización y modificación del escenario, y a partir de él, es posible crear y explotar el escenario.

6.2 CREACIÓN DEL ESCENARIO

Como se ha comentado con anterioridad el constructor de escenarios sirve fundamentalmente para crear escenarios que posteriormente serán utilizados en una tarea robótica. En este apartado se profundiza en las diferentes opciones para crear el escenario, que deberá hacerse inevitablemente utilizando los objetos y robots creados con los otros dos módulos de **RobotScene**, el constructor de objetos, y el constructor de robots.

Añadir objetos

Una parte fundamental en la creación del escenario es poder añadir objetos que posteriormente nuestro robot tendrá la capacidad de manipular. Se pueden añadir objetos al escenario de dos diferentes maneras:

→ *Menú principal > Escena> Nuevo sólido*

→ *Panel Escena > Jerarquía de la escena > Botón añadir objeto*  .

Utilizando cualquiera de estas dos formas el programa nos abrirá la siguiente ventana:

Figura 38. Constructor de escenarios. Ventana para cargar sólidos.

Al pinchar con el ratón en la opción *Cargar objeto* se abrirá un explorador que permite seleccionar objetos almacenados en el disco duro. Los objetos que se carguen deben haber sido creados ineludiblemente mediante el correspondiente constructor de objetos, por tanto, el formato de estos archivos debe ser *.3d*. Una vez observemos en la ventana la geometría del objeto cargado debemos aceptar y el objeto formará ya parte del escenario y se podrá observar en el visor 3D. Cada sólido añadido al escenario tiene asociado un sistema de referencia que debemos tener en cuenta a la hora de crear el escenario.

Aunque no es recomendable, es posible añadir objetos al escenario sin geometría. De esta manera se consiguen puntos en el escenario que pueden ser útiles a la hora de crear una trayectoria por puntos de registro.

Añadir robots

Para que un robot pueda manipular los sólidos que formen parte del escenario debemos incluir al menos un robot en el escenario. Añadir un robot al escenario se realiza de forma similar a añadir un objeto. Podemos añadir un robot al escenario de las siguientes formas:

→ *Menú principal >Escena>Nuevo robot*

→ *Panel Escena > Jerarquía de la escena >Botón añadir robot*  .

Utilizando cualquiera de estos dos procedimientos el programa nos abrirá la siguiente ventana:

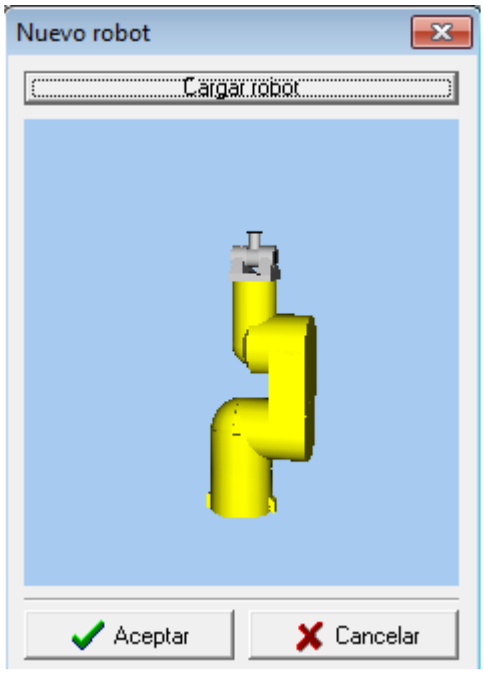


Figura 39. Constructor de escenarios. Ventana para cargar robots.

Si pinchamos con el ratón en la opción *Cargar robot* se abrirá un explorador que permite seleccionar robots del disco duro. Los robots que carguemos han de haber sido creados ineludiblemente mediante el correspondiente constructor de robots, por tanto, el formato de estos archivos debe ser *.rob*. Cuando se observe en la ventana la geometría del robot cargado, se debe y el robot formará ya parte del escenario y aparecerá representado en el visor 3D.

Como ya se ha comentado para los objetos, cada robot añadido al escenario tiene asociado un sistema de referencia que debemos tener en cuenta a la hora de crear el escenario. El sistema de referencia del robot es coincidente con el sistema de referencia absoluto, conocido en robótica como sistema de coordenadas mundo. El constructor de escenarios nos da la posibilidad de trasladar y rotar el robot como pasaba con los objetos, cuando esto ocurra el programa realizará automáticamente el cambio del sistema de referencia mundo. Lo habitual en robótica es cambiar la referencia mundo en el programa robótico mediante la instrucción WORLD.

Añadir una herramienta al robot

Generalmente nuestro robot necesitará de una herramienta para realizar la tarea robótica asignada. Para incluir una herramienta en un robot en el constructor de escenarios, es necesario que se haya cargado en la escena al menos un robot. Se añade una herramienta a un robot accediendo al modo de operación *Añadir herramienta.* Es posible hacerlo de las siguientes formas:

→ *Menú principal >Acción >Añadir una herramienta al robot*

→ *Barra de herramientas > Botón añadir una herramienta al robot.*  .

Una vez que se haya accedido a este modo de operación, se selecciona el robot al cual se desea añadir una herramienta. Para seleccionarlo se pincha sobre el robot en el visor 3D o sobre el nombre del robot en la jerarquía de la escena en el panel de la zona derecha del constructor de escenarios. El programa nos mostrará la siguiente ventana:

Figura 40. Constructor de escenarios. Ventana para añadir herramienta.

Al pinchar con el ratón en la opción *Cargar herramienta* se abrirá un explorador para seleccionar las herramientas del disco duro. Sólo es posible cargar aquellas que han sido creadas previamente mediante el correspondiente constructor de robots; por tanto, el formato de estos archivos debe ser *.herr*. Cuando la ventana muestre la geometría de la herramienta cargada, se acepta y la herramienta formará ya parte del robot y se podrá visualizar en el visor 3D.

Si el robot ya tenía colocada una herramienta al seleccionar el robot en el modo de operación de añadir herramientas, la ventana cargará automáticamente la geometría de la herramienta. Si se desea eliminar la herramienta tan sólo es necesario pulsar el botón *Eliminar herramienta*.

Jerarquía de la escena

Como se ha comentado en la descripción de la interfaz, el constructor de escenarios posee en la zona lateral derecha de la ventana un panel mediante el que se puede modificar el escenario y la jerarquía de la escena. Para ello se utiliza el correspondiente *Panel Escena*:

Figura 41. Contructor de escenarios. Panel escena.

Este panel muestra la jerarquía de la escena de forma arbórea. El primer elemento del panel es el escenario que representa la referencia absoluta. A partir de él colgarán todos los demás elementos del escenario. Mediante el panel pueden añadirse robots, objetos e hijos. Los robots únicamente pueden colgar del escenario y no puede haber ningún objeto que cuelgue de los robots. La jerarquía de la escena puede modificarse arrastrando los objetos. Como es lógico no puede arrastrarse un objeto a otro de sus hijos. Este panel permite cambiar el nombre de los objetos haciendo doble click en el nombre del elemento que queramos renombrar.

La estructura jerarquizada de los elementos del escenario acarrea ciertas consecuencias. A la hora de trasladar los objetos, si se hace mediante el visor 3D se modificará su posición absoluta y la de sus hijos. Si se hace numéricamente mediante el correspondiente botón se modificará la posición con respecto al padre.

Para organizar los diferentes elementos se utilizan los botones:  . Su uso permite cambiar la posición de los elementos pero no altera la jerarquía. Aunque no tenga efectos prácticos directos, es posible de esta manera ordenar los elementos al gusto del programador. Adicionalmente, el programa dibujará en pantalla los elementos en orden vertical y descendente de manera que en algunas ocasiones pueden provocar efectos visuales no deseados; por tanto, puede ser necesaria su recolocación en el árbol.

6.3 EXPLOTACIÓN DEL ESCENARIO

Un escenario robótico se compone al menos de un robot y de varios objetos simples que el robot tendrá capacidad para manipular. Cuando se habla de explotación del escenario se alude a la capacidad de entablar comunicación con el robot para que realice una tarea concreta; de este modo, el robot se moverá por el escenario robótico y estará en condiciones de manipular los objetos del escenario, según las instrucciones que se le hayan dado.

La explotación del escenario en **RobotScene** permite, como en una situación real guiar un robot, realizar un registro de trayectorias o programar una tarea robótica mediante lenguaje textual. Otra de las acciones que se puede llevar a cabo en la parte de explotación del escenario es comunicarse con los robots mediante sus entradas y salidas binarias que pueden ser monitorizadas en todo momento en **RobotScene**, y constituyen la herramienta perfecta para posibilitar la comunicación entre robots.

Guiado de robots

Una de las formas de explotar el escenario y de dotar de movimiento a los robots es mediante la herramienta de guiado. En **RobotScene** esta herramienta intenta emular las funciones básicas que se pueden realizar con la botonera de guiado de un robot industrial. En el módulo constructor de robots ya era posible realizar el guiado del robot, mediante la correspondiente ventana de guiado.

En el constructor de escenarios se puede acceder al modo de operación guiar robots de dos formas:

→ *Menú principal > Acción > Guiar robot*

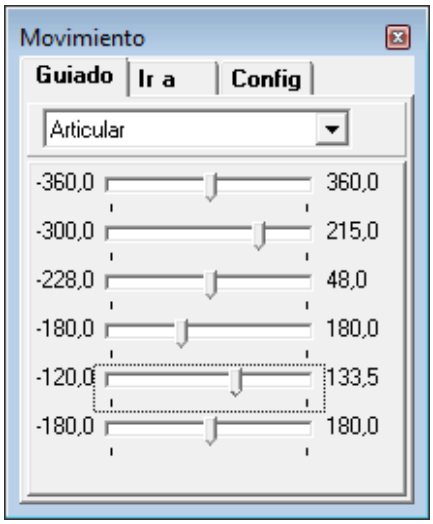
→ *Barra de herramientas > Botón guiar robot*  .

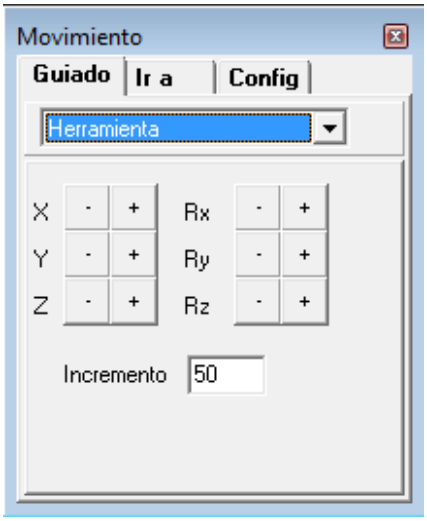
Una vez dentro de este modo de operación basta con pulsar con el botón izquierdo del ratón en el robot que deseamos guiar, o pinchar en su nombre en la jerarquía de la escena. De esta manera se abrirá la ventana de guiado de robots. Una vez abierta la ventana de guiado puede ser utilizada para varios robots, seleccionando el nuevo robot a guiar al pulsar el botón izquierdo del ratón sobre el robot o en la jerarquía de la escena.

La ventana de guiado es una simulación de una botonera de guiado de un robot industrial, por tanto ofrecerá distintas formas de guiado de los robots.

La primera pestaña de la ventana de guiado muestra la función de guiado que puede ser articular o en coordenadas del mundo real.

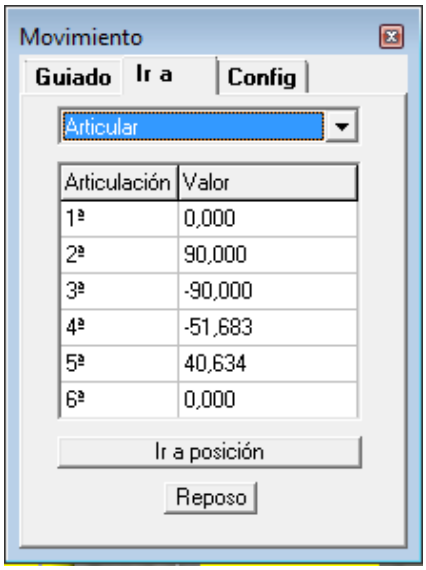
Con el guiado articular es posible guiar, usando el ratón, a cada articulación del robot hacia un valor concreto. En el guiado mediante coordenadas del mundo real es factible guiar el robot - tomando como referencia la absoluta o la de la propia herramienta del robot - aplicando pequeños incrementos en las posiciones y orientaciones. A continuación se muestran las imágenes de la ventana de guiado correspondientes a estos tipos de guiado:

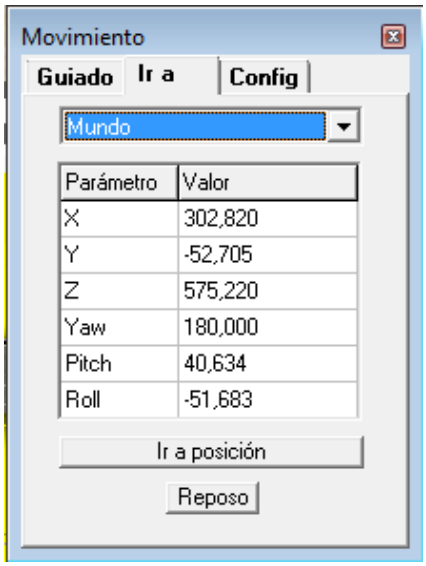




Figuras 43 y 44.- Ventana de guiado: Guiado articular (izquierda) y en coordenadas mundo (derecha)

La segunda de las pestañas se denomina *Ir a.* Su uso permite guiar al robot a una posición y orientación concreta, bien mediante coordenadas articulares o en coordenadas mundo. En esta pestaña también está incluida la opción de ir a las coordenadas de reposo que fueron previamente establecidas en la construcción del robot.





Figuras 45 y 46.- Ir a. Coordenadas articulares (izquierda) e Ir a. Coordenadas mundo (derecha)

Por último la tercera pestaña permite elegir la configuración del robot:

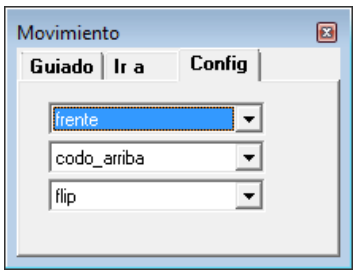


Figura 42. Ventana de Guiado. Pestaña de configuración.

Registro de trayectorias

Una de las formas más básicas de programar una tarea robótica es la del registro de trayectorias. Esta forma de programar consiste en guiar al robot mediante la botonera - ventana de guiado en nuestro caso - hacia puntos concretos que serán almacenados para posteriormente cubrir una trayectoria que el robot será capaz de reproducir. Tiene sus orígenes en las aplicaciones de teleoperación (industria submarina, industria nuclear, etc). Actualmente es utilizado para aplicaciones como la descarga de máquinas o de soldadura por puntos.

Es un método para programar robots fácil y sencillo, para el que no se necesitan conocimientos de informática ni de programación. Sólo basta con habituarse al control del robot mediante la botonera correspondiente e ir registrando una serie de puntos.

Para la implementación del registro de trayectorias en **RobotScene** se decidió utilizar la situación de la herramienta, es decir, su posición y orientación en vez de las posiciones articulares como se suele hacer en la industria robótica. La ventaja de recoger la situación del robot en vez de sus coordenadas articulares es que permite la portabilidad de dicha trayectoria. Si el registro de la trayectoria fuera tomado mediante coordenadas articulares, no se podría utilizar dicha trayectoria en otro robot diferente al que fue creado. En cambio la desventaja que presenta hacerlo de esta manera en un entorno industrial es que el robot tendría poca o nula adaptación a cambios en el escenario.

Para acceder a la ventana que nos permite realizar el registro de trayectorias en **RobotScene**, se debe entrar en el modo de operación de guiado y haber seleccionado previamente un robot. Una vez hecho esto, se activarán las opciones existentes en la aplicación para poder acceder a la ventana de programación. Para poder visualizar la ventana que nos permite realizar la trayectoria por puntos, existen dos maneras:

→ *Menú principal > Ver> Ventana de registro*

→ *Barra de herramientas > Botón ventana de registro*  .

Aparecerá entonces en pantalla la ventana de registro de puntos que se muestra en la siguiente imagen

Figura 43. Ventana de registro de puntos.

Gracias a la ventana de registro de trayectorias es posible programar de forma sencilla una aplicación. Para registrar un punto hay que ir, primeroa la posición dicho punto. Esto lo podemos realizar mediante la ventana de guiado o bien importando la posición del punto que queremos registrar. Para este último

menester la ventana cuenta con una zona denominada *importar* que permite mover el robot a la posición de un objeto perteneciente al escenario. Tan sólo es necesario escribir el nombre del objeto y pulsar en el botón *Ir a* para que el robot se mueva hacia esa posición.

Situados en el punto que se desee registrar, se debe atender a la zona inferior de la ventana denominada *Situación actual*. Se puede registrar el punto de dos maneras diferentes. Mediante *Insertar* se registrará el punto en la posición anterior al punto seleccionado. Mediante *Añadir* se registrará el punto en la última posición de los puntos registrados.

Los puntos registrados se van actualizando en la zona superior de la ventana denominada *Registro de puntos*. En esta zona se puede *Ejecutar* la trayectoria o eliminar todos los puntos pulsando el botón *Reiniciar*.

La zona denominada *Punto seleccionado* permite al robot moverse a la posición del punto que se encuentre seleccionado y que ha debido ser previamente registrado. Desde está zona es posible, también, eliminar cualquier punto si se selecciona y se pulsa el botón *Eliminar*.

Programación textual del robot

Como anteriormente se ha comentado, es posible realizar la programación del robot mediante el registro de trayectorias de forma simple. Otra forma de programación del robot es mediante la codificación textual de una serie de instrucciones que conformarán un programa robótico que, posteriormente, el robot será capaz de ejecutar. Esta forma de programar el robot es más compleja que la del registro de trayectorias pero, en cambio, es mucho más potente que la anterior.

Para poder programar el robot mediante líneas de código o texto se ha implementado en **RobotScene** una ventana de programación. En esta versión multi-robot del simulador robótico se crea automáticamente una ventana de programación por cada robot añadido al escenario.

Para acceder a la ventana de programación es necesario, como es lógico, que el escenario sobre el que se trabaja cuente al menos con un robot. Si se cumple esta condición se activa el modo de operación *programar robots* de dos formas posibles:

→ *Menú principal > Acción > Programar robot*

→ *Barra de herramientas > Botón programar robot*  .

Una vez dentro dicho modo de operación, se visualiza la ventana de programación de cada robot, tras pulsar sobre el robot deseado en el visor 3D o en el nombre del robot en el árbol de jerarquía. Aparecerá la siguiente ventana que corresponde con el editor de programas:

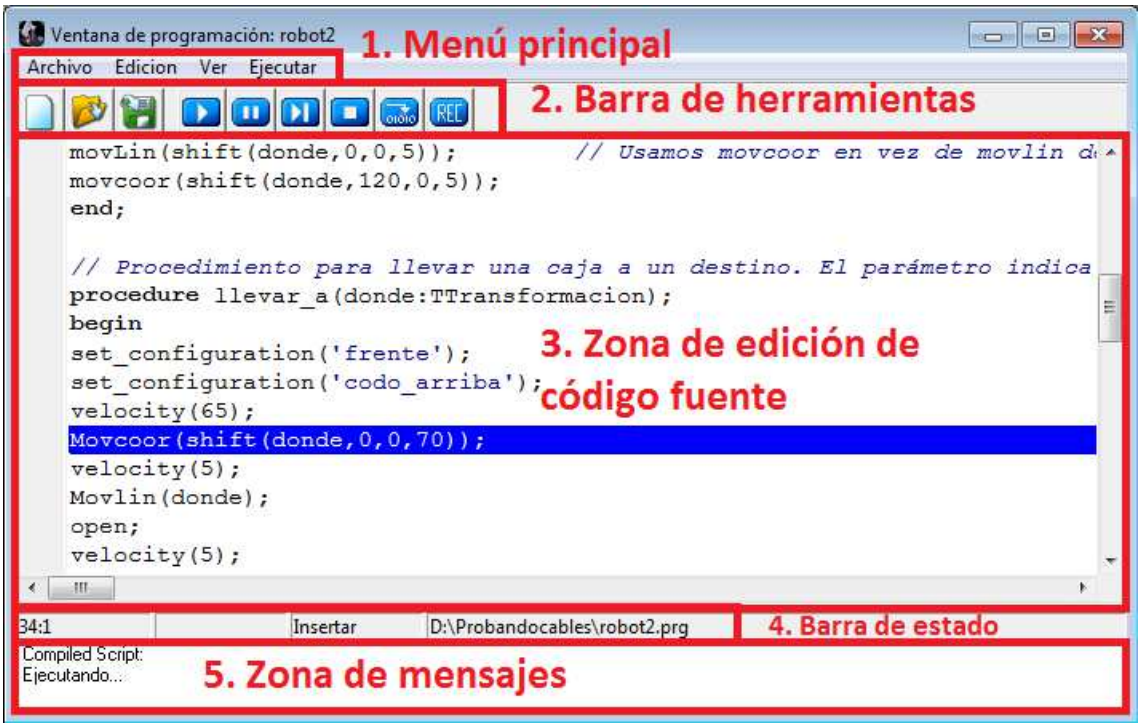


Figura 44. Ventana de programación de robots.

Tal como se observa en la imagen, la ventana de programación consta de las siguientes partes:

1. **Menú principal.**

Desde el menú principal se puede acceder a prácticamente todas las funcionalidades de la ventana de programación. Está compuesto por los siguientes elementos:

Archivo: En esta opción del menú el usuario tiene a su disposición los elementos relativos al manejo de ficheros. *Nuevo* permite al usuario generar un archivo *.prg* en blanco. *Abrir* permite visualizar al usuario un archivo *.prg* previamente guardado. *Guardar como* permite guardar en un fichero el programa actual, la aplicación permitirá al usuario elegir el nombre y destino del fichero. *Guardar* permite guardar automáticamente el archivo *.prg* con el nombre actual del fichero.

Edición: Desde este submenú el usuario puede realizar las acciones relativas a la edición de texto. *Deshacer* permite volver atrás en los cambios realizados en el editor de código fuente. *Rehacer* elimina el último deshacer realizado. A continuación aparecen una serie de comandos que permiten realizar acciones sobre el texto que son comunes para todos los editores de texto como son: *Copiar*, *Cortar*, *Pegar*, *Eliminar* y *Seleccionar todo.*

Ver: Este submenú permite al usuario acceder a las opciones gráficas de la ventana de programación. *Fuente* permite elegir el tipo de letra, tamaño y color del editor de código. *Ver números de línea* muestra en el editor a la izquierda de cada línea, el número de línea correspondiente. *Ir a línea* abre un cuadro de texto que nos permite escribir el número de línea al que queremos ir de forma automática.

Ejecutar: Desde este submenú se puede acceder a todas las opciones relativas a los modos de ejecución del programa robótico: *Compilar aplicación*, *Ejecutar instrucción*, *Ejecutar aplicación*, *Detener ejecución*, *Pausa* y *Grabar en video*. Estos modos de ejecución del programa se explicarán con más detalle más adelante.

2. Barra de herramientas

La ventana de programación del constructor de escenarios cuenta con una barra de herramientas que permite realizar todas las acciones de la ventana.

Figura 45. Barra de herramientas de la ventana de programación.

Las acciones que realizan cada uno de los botones son

 Crea un nuevo archivo de programa.

 Carga un archivo de programa previamente guardado.

 Guarda el archivo de programa actual.

 Compila y ejecuta el programa robótico actual.

 Pausa la ejecución del programa.

 Ejecuta una instrucción del programa.

 Detiene la ejecución del programa.

 Compila el programa.

 Ejecuta el programa robótico guardando la imagen del visor 3D en un archivo de video.

3. Zona de edición de código fuente

Esta es la zona reservada para que el usuario pueda escribir el código fuente de la tarea robótica. Posteriormente dicho código será compilado y ejecutado por un robot. La sintaxis del programa robótico es similar a Pascal y será estudiada con mayor detenimiento en siguientes capítulos.

El editor de texto resaltará las palabras clave del lenguaje de programación (if, then, else, while, etc), facilitando la lectura del código y su compresión. Los comentarios en el código fuente, aparecerán en azul y en cursiva para mejorar su visualización.

Cuando la tarea robótica se encuentre en ejecución el editor de código remarcará con fondo azul la línea que se está ejecutando en ese momento.

4. Barra de estado

La barra de estado muestra información relativa al editor de texto. Está dividida en cuatro casillas que muestran diferente información. La primera casilla nos indica la posición del cursor en el editor en formato fila:columna. La segunda de las casillas muestra si el archivo ha sido modificado desde su apertura. La tercera casilla nos muestra la opción actual de introducción de texto en el editor podrá ser insertar o sobreescribir. Por último la cuarta casilla se rellena con la ruta del archivo abierto.

5. Zona de mensajes

En esta zona del editor se muestran los mensajes de notificación del compilador. Es la vía que utiliza el compilador para avisarnos de lo que ha sucedido. En este lugar aparecerán mensajes avisando de errores en la compilación o del estado de ejecución del programa. Por ejemplo nos avisará si se ha compilado, ejecutado o pausado un programa robótico.

Tipos de ejecución del programa robótico

Como se ha podido observar en la definición de los controles de la ventana de programación, al ejecutar una tarea robótica **RobotScene** cuenta con diferentes tipos de ejecución y controles que capacitan al usuario para el completo control de ejecución del programa robótico.

Compilar programa: Mediante esta acción se genera un código interno que será utilizado posteriormente por el motor de scripts para la ejecución de la tarea robótica. No es una compilación en el sentido estricto de la palabra puesto que no se genera ningún archivo ejecutable en código máquina. Existen tres maneras diferentes para realizar la compilación del código robótico:

→ *Menú principal > Ejecutar> Compilar aplicación.*

→ *Barra de herramientas > Botón compilar la aplicación*  .

→ *Combinación de teclas: Ctrl + F9.*

Al realizar esta acción, en la zona de mensajes de la ventana de programación aparecerá la información relativa a la compilación que avisará si el programa ha sido compilado con éxito o si el código contenía algún error sintáctico, remarcando la línea errónea de color rojo.

Ejecutar una instrucción: Permite la ejecución de una sola línea de programa siguiendo el flujo de programa actual. Tras ejecutar la instrucción el script se mantendrá pausado (modo pausa) a la espera de nuevas acciones de control del programa. Si en el momento de realizar la acción no esta en ejecución el programa robótico se realizará una compilación previa. Se puede ejecutar una instrucción mediante tres caminos:

→ *Menú principal > Ejecutar> Ejecutar instrucción.*

→ *Barra de herramientas > Botón ejecutar instrucción*  .

→ *Tecla F8.*

También conocido como ejecución paso a paso, este modo de control de programa es muy útil para mantener el control del flujo de ejecución del programa y para labores de depuración. Este modo de control facilita la detección y resolución de errores en el código del programa. Para facilitar la visualización de la línea actual en ejecución, se remarca el fondo de dicha línea de color azul.

Ejecutar la aplicación: Este modo de ejecución es el más utilizado. En este modo, el script ejecutará instrucciones de programa sucesivamente hasta finalizar todas las instrucciones. Este modo realiza una compilación previa para detectar errores en el código de la tarea. El editor de código remarca la línea actual en ejecución con color azul. Para realizar la ejecución completa de la aplicación se pueden utilizar tres opciones diferentes:

→ *Menú principal > Ejecutar > Ejecutar aplicación.*

→ *Barra de herramientas > Botón ejecutar aplicación*  .

→ *Tecla F9.*

Detener la ejecución: Detiene por completo la ejecución del script. La próxima vez que se ejecute la aplicación o una instrucción, lo hará desde la primera línea de código y previa compilación del código. Para acceder a esta opción son válidas cualquiera de las siguientes alternativas:

→ *Menú principal > Ejecutar> Detener ejecución.*

→ *Barra de herramientas > Botón detener ejecución*  .

→ *Combinación de teclas Ctrl + F2.*

Pausa: Este modo de ejecución pausa la ejecución del script de forma que pueda recuperarse el flujo de ejecución del programa posteriormente. La principal diferencia con el modo *detener ejecución* es que permite recuperar el flujo de programa cuando la aplicación vuelva a reanudarse, ya sea mediante la ejecución de la aplicación o mediante una ejecución paso a paso. Para pausar el script podemos realizar las siguientes acciones:

→ *Menú principal > Ejecutar> Pausa.*

→ *Barra de herramientas > Botón pausa*  .

Grabar en video: Mediante este modo se puede ejecutar la aplicación mientras el escenario robótico está siendo grabado en video. La única diferencia entre este modo de ejecución y el de ejecutar la aplicación es que éste genera, durante la ejecución de la tarea, un archivo *.avi* en el que se almacena el video de la escena. Para establecer este modo de ejecución se nos presentan dos opciones:

→ *Menú principal > Ejecutar > Grabar en video.*

→ *Barra de herramientas > Botón grabar en video*  .

Como ocurría al ejecutar la aplicación, se realizará una compilación previa. Si esta compilación es exitosa se abrirá un explorador para guardar el video con el nombre deseado. A continuación se abrirá otra ventana que permitirá elegir el codec (vid, Mpeg4, DivX, etc) de compresión y otras opciones como la calidad de compresión o la velocidad de datos. El video se grabará a unos 25 FPS.

Modelado del entorno

Cabe recordar en este apartado que, tanto para la construcción del escenario como para su explotación, se utiliza la misma interfaz. Sin embargo el modelado del entorno es diferente para cada modo de utilización del escenario.

En la parte de creación del escenario es posible definir relaciones jerárquicas entre los diferentes objetos que componen el escenario. Por su parte, los objetos cuentan con unas propiedades tales como posición, orientación, geometría y relaciones jerárquicas. En cambio, cuando se explota el escenario, los objetos sólo cuentan con posición y orientación dentro de sus propiedades.

Comunicación entre robots

Uno de los requisitos principales de la versión multi-robot de **RobotScene** es que los robots tuvieran capacidad para comunicarse entre sí. Para lograr este requisito se ha implementado en **RobotScene** una ventana que pretende simular el cableado existente en instalaciones industriales donde intervienen varios robots de forma simultánea. Por lo tanto, la ventana de cableado posibilita la creación de un cableado entre robots de una manera sencilla, intuitiva y visual, que será guardado en un archivo con formato *.cbl*.

Para la realización del cableado se ha simulado una botonera de 10 entradas y 10 salidas para cada robot.

Para poder acceder a la ventana de cableado es imprescindible que el escenario disponga de al menos dos robots. Si han sido añadidos a nuestra escena al menos dos robots se activarán automáticamente las opciones para acceder a dicha ventana. Desde el constructor de escenarios se puede acceder a la ventana de cableado de dos formas diferentes:

→ *Menú principal > Ver > Ventana de cableado.*

→ *Barra de herramientas > Botón ventana de cableado*  .

Accediendo de cualquiera de las maneras se mostrará en pantalla la ventana de cableado que se presenta del siguiente modo:

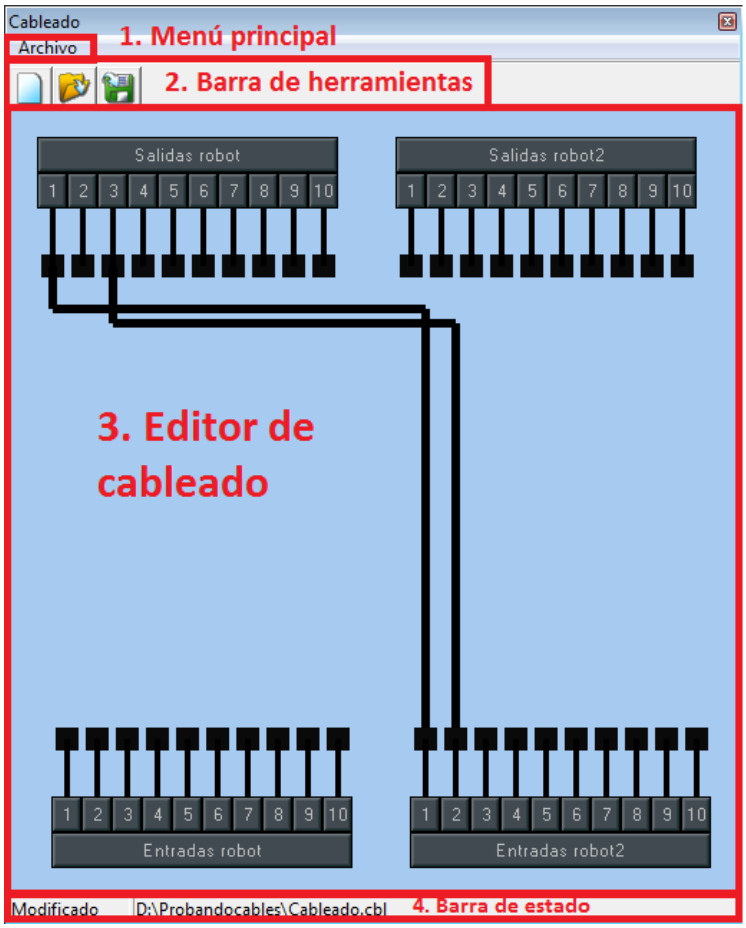


Figura 46. Constructor de escenarios. Ventana de cableado.

Tal como podemos observar en la imagen la ventana de cableado presenta las siguientes partes:

1. **Menú principal.**

Desde el menú principal podemos acceder a a las opciones realativas al guardado y carga del escenario.

Archivo: Esta es la única opción del menú, en ella el usuario tiene a su disposición los elementos relativos al manejo de ficheros.

Nuevo permite al usuario generar un archivo *.cbl* en blanco. Es posible acceder a esta opción mediante la combinación de teclas Ctrl + N.

Abrir permite visualizar al usuario un archivo *.cbl* previamente guardado. Es posible abrir un archivo mediante la combinación de teclas Ctrl + A.

Guardar como permite guardar en un fichero el cableado actual, la aplicación permitirá al usuario elegir el nombre y destino del fichero.

Guardar permite guardar automáticamente el archivo *.cbl* con el nombre actual del fichero. Es posible guardar el archivo mediante la combinación de teclas Ctrl + G.

Cerrar ventana, cierra la ventana de cableado.

2. Barra de herramientas

La barra de herramientas permite realizar las acciones relativas al manejo de ficheros de cableado.

 Crea un nuevo archivo de cableado.

 Carga un archivo de cableado previamente guardado.

 Guarda el archivo de cableado actual.

3. Editor de cableado

Es la parte más importante de la ventana de cableado. Permite al usuario editar un archivo de cableado que simula un cableado real. Esta ventana ajusta su tamaño al número de robots en el escenario. Como se puede observar en la imagen de la ventana, en la zona superior se encuentran las botoneras que reúnen las salidas binarias de los robots industriales. Por otro lado, en la parte inferior se encuentran las botoneras correspondientes a las entradas digitales de los robots.

Este editor permite cablear las salidas de un robot con las entradas de los demás robots del escenario. Para realizar dicho cableado se pone a disposición del usuario esta interfaz en la que se pueden simular cableados de forma gráfica e interactiva. A continuación se describe el modo de uso del editor.

Creación de un cable

La creación del cable debe hacerse de forma interactiva entre el ratón y el editor. El usuario puede mover el ratón por el editor. Si el usuario coloca el cursor encima de cualquiera de los pads de salida, observará que dicho pad cambia a color verde, lo que representa que es posible comenzar la creación del cable.

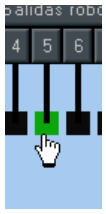


Figura 47. Cursor sobre un pad de salida en la ventana de cableado.

Para crear el cable solamente es necesario pulsar sobre un pad de salida y arrastrar el ratón manteniendo pulsado el botón izquierdo del ratón. De esta manera se ha creado un cable que puede ser arrastrado por la pantalla.

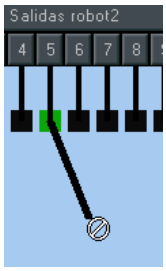


Figura 48. Ventana de cableado. Creación del cable.

Como se observa en la imagen anterior, el cursor cambia su imagen a una señal de prohibido que indica que, para terminar con la creación del cable, no es posible soltar el ratón en ese punto. También se advierte que los pads de entrada en los que puede terminar el cable se colorean en verde.

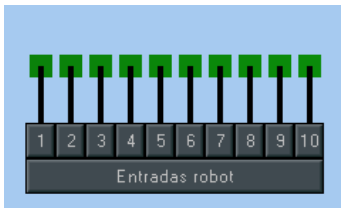


Figura 49. Ventana de cableado. Los pads de entrada válidos para crear un cable se colorean en verde.

Finalmente, para terminar con la creación del cable solo es necesario llevar el cursor del ratón a un pad de entrada válido y, a continuación, soltar el botón izquierdo del ratón. Para asegurarnos de que el pad de entrada es válido, debemos observar que el pad es de color verde y que cambia la imagen del cursor.

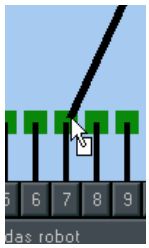


Figura 50. La imagen del cursor nos informa de que es un pad de entrada válido.

De esta manera se habrá creado un cable entre el pad de salida y el pad de entrada deseados. El programa implementará internamente dicho cable de modo que será posible la comunicación entre robots utilizando instrucciones propias del lenguaje robótico como ***wait***, o ***signal***.

Eliminación de un cable

El editor de cableado, permite a su vez que se puedan eliminar cables. Eliminar un cable en el editor es tan sencillo como situar el cursor del ratón encima del cable que se desea eliminar y pulsar el botón derecho del ratón.

Monitorización de entradas y salidas

RobotScene cuenta con una ventana que permite monitorizar de manera simple las entradas y salidas binarias de los robots del escenario. La ventana de monitorización se modifica automáticamente de acuerdo con el número de robots del escenario. Para poder visualizar la ventana de monitorización es imprescindible que el escenario cuente con al menos un robot. Si el escenario cuenta con uno o más robots se activarán las opciones que permiten al usuario abrir la ventana. Dichas opciones son dos y se puede acceder a ellas mediante

→ *Menú principal > Ver > Ventana E/S.*

→ *Barra de herramientas > Botón ventana de E/S* E/S.

Con cualquiera de las dos opciones se mostrará en pantalla la ventana de entradas y salidas que se corresponde con la siguiente imagen:

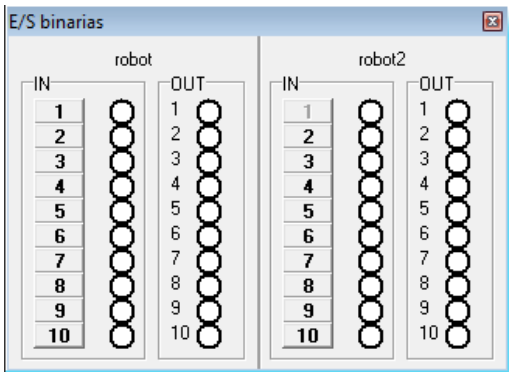


Figura 51. Constructor de escenarios. Ventana E/S.

La función de la ventana es la monitorización de las entradas y salidas; sin embargo, simulando una botonera de entradas en un robot real, las entradas son interruptores de tres estados.

Los dos primeros estados se corresponden con el 1 y el 0 binarios. El tercero de los estados se corresponde con entrada inhabilitada, que en la actual aplicación equivale a que dicha entrada se encuentra cableada. La inhabilitación de las entradas se realiza de forma automática cuando se cablea una entrada. En la botonera de entradas se ha implementado un botón por cada entrada que permite poner a 1 o 0 el valor de la entrada. En las siguientes imágenes se pueden observar el estado de las entradas:

Figura 52. Ventana E/S. Entrada con valor 0, con valor 1 y deshabilitada.

Adicionalmente, cabe destacar que es posible abrir la ventana de monitorización durante la ejecución de un programa robótico por lo que constituye una buena herramienta para conocer el estado de entradas y salidas. Cuando una salida cambia su estado al 1 binario se observará de color rojo, como puede

apreciarse en la siguiente imagen en la que se activa, la salida 1 del robot 1 y la entrada 1 del robot 2 que previamente han sido cableadas:

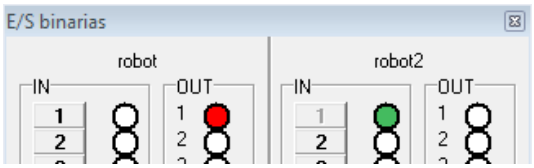


Figura 58. Una salida y una entrada cableadas se señalan de diferente color para mejorar su visualización.

6.4 ESTRUCTURA DEL CÓDIGO FUENTE

En este apartado se explica la manera en la que el código fuente del simulador trasladará los objetos que componen el escenario, los robots que participan en la simulación, el manejo de las herramientas y, en definitiva, la forma de establecer la comunicación entre los robots. Además, para que esas acciones puedan ser visualizadas en la pantalla, es necesario definir e implementar aquellas funciones, clases, eventos y procedimientos que las hagan posibles

En el constructor de escenarios, el código de la parte relativa al manejo de las vistas y la cámara es prácticamente el mismo que en otros módulos. En cuanto a la parte relativa a la construcción del escenario, hay que comentar que se realiza de forma similar en el constructor de objetos puesto que las acciones son las mismas; por ejemplo acciones de rotación y translación.

En el constructor de escenarios tendremos dos tipos de objetos diferentes, los robots y objetos simples.

1. Clase TObjeto_escena.

Esta clase está definida en la unidad solido.pas. Como se ha comentado, a nivel de código fuente, en el constructor de escenarios sólo existirán dos clases diferentes de objetos. Por tanto, se utilizará la clase TObjeto_escena para referirse a cualquier objeto inanimado que forme parte del escenario. La clase cuenta con una geometría del tipo TSolido y con los métodos necesarios para la correcta utilización de esta clase, como pueden ser aquellos que permiten cargar los objetos en el escenario y la representación de sus ejes. A continuación se puede observar el código relativo a esta clase.

```
TObjeto_escena = class(TGLCustomSceneObject)
  private
    Fejes:boolean;
    ejes:TGLDummyCube;
    alojamiento_ejes:TGLDummyCube;
    procedure setEjes(const valor:boolean);
    procedure crear_ejes;

  public
    nodo:TTreeNode;
    nombre:string;
    geometria:TSolido;
    property ejes_visibles:boolean read fejes write setEjes;
    procedure iniciar;
    procedure cargar(archivo:string);
    procedure marcar;
    procedure desmarcar;
  end;
```

La representación de los ejes se realiza mediante las diferentes clases que nos facilita GLScene. Para la representación de los ejes en estos objetos se ha implementado un flag (ejes_visibles) que indica si deben estar visibles u ocultos.

Como se puede observar en el código, la clase utiliza una serie de variables para guardar en memoria el nombre del objeto y su posición en la estructura arbórea de la jerarquía.

2. Manipulación del escenario virtual.

Para manipular los elementos del escenario virtual se ha implementado en el constructor de escenarios una función parecida a la ya existente en el constructor de objetos. La función *coger_solido* nos devolverá el objeto de tipo *TSolido* que se encuentre bajo el puntero del ratón.

```
function coger_solido(x,y:integer):TSolido;
```

Para conocer el tipo de objeto seleccionado con el ratón se ha implementado la propiedad *tipo*. Esta propiedad tiene tres valores posibles de_robot, de_escenario, de_herramienta.

Para manipular los objetos del escenario, se ha añadido al constructor de objetos dos variables globales: *solido_arrastrado* y *solido_activo*, del tipo *TGLCustomSceneObject*, que permite tratar los objetos seleccionados independientemente de si son robots u objetos inanimados. Los campos geometría, posición y orientación de la interfaz de usuario trabajarán sobre el solido activo, mientras que *solido_arrastrado* esta pensado para trabajar con el objeto que este siendo arrastrado mediante la utilización del botón izquierdo del ratón.

Para la manipulación de los objetos que forman parte del árbol jerárquico se deben utilizar las propiedades absolutas. Sin embargo, GLScene trabaja con las propiedades relativas al antecesor del objeto; por esta razón se mueve a la raíz de la escena el objeto que se quiera manipular y. una vez manipulado. el objeto es devuelto a su posición original en el árbol. Para ello se utilizan los siguientes procedimientos.

```
padre:TObjeto_escena;
 …
 function tiene_padre(objeto:TGLCustomSceneObject):boolean;
 procedure enviar_a_raiz;
 procedure enviar_a(donde:TObjeto_escena);
```

Estos procedimientos se utilizan en los eventos del visor 3D.

En el evento *OnKeyDown* se debe comprobar si el objeto está o no en la raíz, es decir si tiene padre o no:

```
…
 if tiene_padre(solido_activo) then
    enviar_a_raiz;
```

Mientras que en el evento *OnMouseUp* se añaden las siguientes líneas de código:

```
…
 if (solido_arrastrado<>NIL)and(padre<>NIL) then
    enviar_a(padre);
 padre:=NIL;
…
solido_arrastrado := NIL;
…
```

3. Inclusión de un componente de tipo árbol (TTreeView).

El componente *TTreeView* es el que proporciona *Delphi* para el manejo de árboles. Cada componente o nodo del árbol va encapsulado bajo la clase *TTreeNode*, de manera que es posible establecer el nombre del nodo o elemento mediante la propiedad *text*. La propiedad *ImageIndex* permite añadir imágenes a los nodos. *SelectedIndex* permite establecer imágenes para cada nodo cuando se encuentran seleccionados. Para la gestión de cada elemento utilizado en el árbol son necesarias ciertas funciones. Por ejemplo, *MoveTo* permite mover un elemento a cualquier posición del árbol y *AddChild* permite añadir hijos a los elementos del árbol.

En nuestra aplicación es necesario que el árbol además de contener la información jerárquica de los elementos del escenario haga referencia a dichos elementos en el escenario para poder trabajar con ellos y que, además, los cambios realizados afecten tanto a la representación en el *TreeView* como a la representación del objeto en el escenario tridimensional. Para relacionar cada elemento del árbol con el elemento de la escena se ha implementado una clase propia descendiente de la clase *TTreeNode*.

```
TNodo =  class(TTreeNode)
  public
    objeto:TGLCustomSceneObject;
    tipo:(nodo_robot,nodo_objeto,nodo_raiz);
  end;
```

Mediante el campo *objeto* se hará referencia al objeto correspondiente en la escena. El campo *tipo* hace referencia al tipo de objeto, que puede tomar el valor *nodo_robot* cuando estemos haciendo referencia a un robot, *nodo_objeto* cuando estemos haciendo referencia a un objeto simple y *nodo_raiz* cuando nos refiramos al escenario.

Para poder implementar los nodos personalizados al componente TTreeView es necesario utilizar el evento *OnCreateNodeClass* del TreeView, evento que se ejecuta cada vez que se crea un nuevo nodo:

```
procedure TForm1.TreeView1CreateNodeClass(Sender: TCustomTreeView;
                var NodeClass: TTreeNodeClass);
begin
  nodeclass:=Tnodo;
end;
```

TTreeView permite el arrastre de un elemento de una parte del árbol a otra, pero para adecuarlo a nuestra aplicación, ha sido necesario añadir ciertas

restricciones: no es posible colgar ningún objeto del robot. Como es lógico, tampoco se permite arrastrar un objeto a uno de sus hijos.

Para que un cambio en la jerarquía de la escena a través del árbol se vea reflejado en el escenario de forma automática, se debe añadir cierto código en el evento *OnDragDrop* del árbol. Dicho evento se activa cuando se está arrastrando un elemento en el árbol y se suelta el botón del ratón.

```
Var
  Nodo_Destino,Nodo_Origen:TNodo;
  solido:TObjeto_escena;
Begin
  ...
  solido:=nodo_origen.objeto as TObjeto_escena;
  Nodo_Origen.MoveTo(Nodo_Destino,naAddChildFirst);
  solido.MoveTo(nodo_destino.objeto as TObjeto_escena)
  ...
end;
```

4. Explotación del escenario. Clase TRobot.

En este apartado se examina el código relativo a las rutinas referentes al movimiento del robot, la adición de herramientas y la posibilidad de manipular el entorno mediante el uso de la herramienta.

El movimiento del robot puede ser analizado desde dos perspectivas diferentes. La primera de esas alternativas es tratar al robot como un sistema de eventos discretos, donde las transiciones entre estados se producen a través de eventos. La segunda alternativa es tratar al robot como un sistema continuo.

Si se atiende al robot como un sistema de eventos discretos, se puede observar que el robot puede encontrarse en varios estados internos. El estado interno del robot se almacena en el campo *estado* de la clase *TRobot*.

```
estado:(parado,movimiento_articular,movimiento_rectilineo,
        espera, accionando_herramienta, espera_entrada);
```

Los estados que puede tomar la variable *estado* se definen a continuación.

parado: El robot se encontrará en este estado cuando esté inactivo, es decir, cuando no exista movimiento ni actividad en él. El robot se considera *parado* cuando no se esté ejecutando ningún programa robótico sobre el robot, ni se esté guiándolo mediante la ventana de guiado.

movimiento_articular: El robot presenta este estado cuando se encuentra realizando un movimiento coordinado articular, movimiento que describiremos más adelante. Una vez que el robot haya terminado el movimiento, el estado interno pasará a ser *parado;* en ese momento se analiza si se debe ejecutar otra instrucción de movimiento, o bien terminar la ejecución del programa.

movimiento_rectilineo: El robot presenta este estado cuando se encuentra realizando un movimiento en línea recta, movimiento que describiremos más adelante. Una vez que el robot haya terminado el movimiento, el estado interno pasará a ser *parado*, en ese momento se analiza si se debe ejecutar otra instrucción de movimiento, o bien terminar la ejecución del programa.

espera: El robot presenta este estado cuando se ejecutan sobre él instrucciones para mantener al robot inactivo durante una duración de tiempo limitada. Por ejemplo, el robot estará en *espera* cuando se haya ejecutado sobre él una instrucción del tipo *pause*. Cuando pase el tiempo especificado el robot pasará al estado *parado*.

accionando_herramienta: El robot se encontrará en este estado en los momentos en los que el robot esté abriendo o cerrando la herramienta que lleve asociada. Una vez que haya terminado el movimiento de la herramienta pasará al estado *parado*.

espera_entrada: El estado del robot tomará este valor cuando se encuentre esperando a que se active o desactive una de sus entradas. Este caso se presenta cuando se ejecutan sobre él instrucciones del tipo *wait*. Del mismo modo que en los anteriores casos, una vez cumplida la condición, la variable *estado* tomará el valor *parado*.

Hay que tener en cuenta que se han incluido otras variables booleanas utilizadas para controlar el estado de robot. Como el flag *recien_parado*, que se activa nada más terminar el movimiento, de forma que se pueda conocer información sobre el estado del robot desde fuera de la clase.

5. Movimiento coordinado articular.

El movimiento que un robot debe realizar para moverse desde un punto inicial a un punto final puede ser realizado según infinitas trayectorias espaciales. Si conocemos las coordenadas articulares del punto inicial y las coordenadas articulares del punto final, la estrategia más sencilla será realizar una interpolación lineal entre los valores iniciales y los finales para cada articulación, lo que se conoce como interpolación articular.

El movimiento articular podría definirse como el movimiento que debe realizar un robot para trasladarse desde una posición y orientación concreta a otra, de manera que las articulaciones del robot adecuen su movimiento para que todas las articulaciones lleguen a su posición angular final en el mismo instante de tiempo.

Para explicar el método de resolución de este movimiento supongamos que se pueden desarrollar velocidades constantes entre las posiciones articulares en el momento inicial $q(k)$ y las posiciones en el instante final $q(k+1)$. La duración del movimiento a máxima velocidad para la articulación *i* vendrá dada por:

$$\delta_i = \frac{\Delta q_i}{V_i} = \frac{q_i(k+1) - q_i(k)}{V_i}$$

Donde Vi representa la máxima velocidad admisible por cada articulación.

Al ser un movimiento articular coordinado, todas las articulaciones deben terminar su movimiento a la vez, por lo que la articulación que más tiempo invierta en la evolución será la articulación que defina el tiempo T del movimiento:

$$T = máx(\delta_i)$$

Si para una determinada articulación se ha obtenido que $\delta i \geq T$ la velocidad deseada para dicha articulación no será la máxima, por lo que la i-ésima articulación deberá evolucionar a una velocidad dada por:

$$v_i = \frac{\Delta q_i}{T}$$

En este planteamiento se ha supuesto que al menos una de las articulaciones evoluciona a la máxima velocidad; sin embargo, el constructor de robots permite, por medio del lenguaje robótico, ajustar la velocidad a un porcentaje de la velocidad máxima. Por tanto, el cálculo de tiempo del movimiento se realizará mediante la siguiente expresión:

$$\delta_i = \frac{\Delta q_i}{V_i} = \frac{q_i(k+1) - q_i(k)}{V_i \max - \frac{Velocidad}{100}}$$

Para la implementación de este movimiento en el simulador se han utilizado una serie de variables globales.

```
// Variables internas para el movimiento
//movimiento coordinado
q_inicial:Tarray;      //Valores iniciales de las articulaciones
q_final:Tarray;        //Valores finales de las articulaciones
//movimiento en línea recta
Transformacion_inicial,Transformacion_intermedia:TTransformacion;
Transformacion_final:TTransformacion;
p:TPar_Rotacion;
// Velocidades para la interpolación articular;
velocidades:Tarray;
// Velocidades para la interpolación cartesiana;
velocidades2:record
              x,y,z,tita:single;
             end;
T:double;            // Tiempo de duracion del movimiento
t_actual:double;     // Tiempo transcurrido del movimiento
```

Los vectores *q_inicial* y *q_final* contienen, respectivamente los valores articulares iniciales y finales del movimiento. Las velocidades a las que se moverá cada articulación y que tendremos que calcular están almacenadas en otro vector denominado *velocidades*. Para el control del tiempo, disponemos de la variable *T*, en la que almacenaremos la duración del movimiento previamente calculada, y la

variable *t_actual*, que nos indicará el tiempo transcurrido desde el inicio del movimiento.

Si el destino del movimiento sólo se conoce en coordenadas de la tarea, se debe resolver previamente el modelo inverso de modo que se traduzca el destino en coordenadas de la tarea a coordenadas articulares.

Mediante el siguiente procedimiento se realizan los cálculos necesarios para que el robot este en condiciones de realizar el movimiento.

```
procedure TRobot.mover_coordinado(destino:TArray);
```

El cálculo de tiempo del movimiento se puede obtener de la siguiente manera:

```
T:=0;
for i:=1 to numero_articulaciones do
begin
  articulacion:=articulaciones[i] as TArticulacion;
  v:=articulacion.v_max*velocidad/100;
  delta:=abs((q_final[i]-q_inicial[i])/v);
  if delta>T then
  T:=delta;          // El tiempo es el máximo de todas las deltas
end;
```

Para calcular las velocidades concretas de cada articulación se utiliza la siguiente expresión:

```
// Se calculan las velocidades de cada articulación
for i:=1 to numero_articulaciones do
  velocidades[i]:=(q_final[i]-q_inicial[i])/T;
```

En este paso, el simulador habrá calculado las velocidades articulares y habrá actualizado el estado del robot a *movimiento_articular.* Por lo tanto, en el siguiente disparo del *cadencer* se realizará la interpolación y el cálculo de las variables articulares por medio del procedimiento *mover* de la clase *TRobot* al que se le traspasa, como parámetro, el tiempo transcurrido entre el anterior disparo del cadencer y el actual, dicho tiempo se denomina *deltatime*.

```
procedure TForm1.GLCadencer1Progress(Sender: TObject; const deltaTime,
  newTime: Double);
begin
// Tratamiento de las 'entradas' de la interfaz o del programa robótico
...
Robot.mover(deltatime);
...
end;
```

En el procedimiento *mover* de la clase *TRobot* se realiza la interpolación.

```
procedure TRobot.mover(tiempo:double);
...
if estado=movimiento_articular then
 begin
  t_actual:=t_actual+tiempo;
```

```
  for i:=1 to numero_articulaciones do
  begin
    articulacion:=articulaciones[i] as TArticulacion;
    if t_actual<T then          // Se está moviendo
      articulacion.valor:=articulacion.valor+velocidades[i]*tiempo
    else
    begin                       // Ha terminado de mover
      articulacion.valor:=q_final[i];
      estado:=parado;
      recien_parado:=true;
    end;
  end;
end
```

6. Cambio de referencia de la base y de la herramienta.

Para poder calcular el movimiento relativo a los robots es necesario conocer el destino final de dicho movimiento. En algunas ocasiones el destino se especifica en relación al sistema de referencia mundo, y sin embargo, el modelo inverso se calcula a través de la relación de sólidos en el robot, desde el último de los sólidos a la base. Por tanto, será necesario realizar una serie de transformaciones que permitan traducir el destino respecto del sistema de referencia de la herramienta al sistema de referencia mundo, para poder de esta manera calcular el modelo inverso.

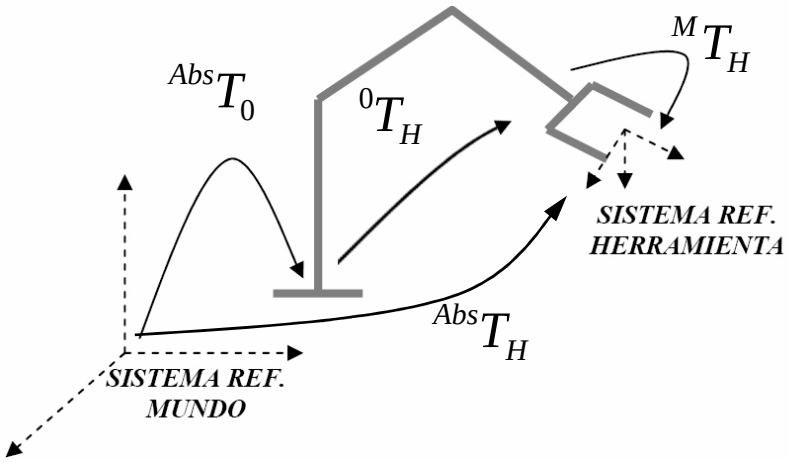


Figura 53. Relaciones entre transformaciones.

Como puede observarse en el gráfico anterior, el destino es especificado mediante programa o mediante guiado a través de la transformación ${}^{ABS}T_H$, la transformación que resuelve el modelo inverso es ${}^{0}T_H$, y la transformación que relaciona el sistema de referencia mundo y la base del robot es ${}^{ABS}T_0$. De esta forma se puede obtener la expresión del modelo inverso:

$$^{Abs}T_H = {}^{Abs}T_0 * MI * {}^{M}T_H$$

$$MI = {}^{Abs}T_0^{-1} * {}^{Abs}T_H * {}^{M}T_H^{-1}$$

Para trabajar con estas transformaciones se ha implementado en **RobotScene** una serie de rutinas para realizar la inversión y multiplicación de matrices que descienden de las librerías de GLScene para el uso de matrices, están definidas en la unidad *algebra.pas*. Se ha decidido implementarlo de esta manera porque así es posible crear tipos propios de matrices, de forma que su uso sea más sencillo e intuitivo.

Las funciones que se utilizarán para este tipo de operaciones serán las siguientes:

```
// Conversión de nuestros tipos a los de GLScene y viceversa
function matrix_a_transformacion(M:TMatrix):TTransformacion;
function transformacion_a_matrix(T:TTransformacion):TMatrix;

// Funciones algebráicas
function Multiplicar(T1,T2:TTransformacion):TTransformacion;
function Invertir(T:TTransformacion):TTransformacion;
```

De esta forma, para un destino dado por la transformación destino, de tipo TTransformación, para resolver el movimiento coordinado deberemos proceder de la siguiente manera:

```
procedure TRobot.mover_coordinado(destino:TTransformacion);
Var
   donde_robot:TTransformacion;
   donde_herramienta:TTransformacion;
…
Begin
 donde_robot:=matrix_a_transformacion(matrix);
 destino:=multiplicar(invertir(donde_robot),destino);
 if herramienta<>NIL then
 begin
   donde_herramienta:=matrix_a_transformacion(herramienta.transformacion.Matrix);
   destino:=multiplicar(destino,invertir(donde_herramienta));
 end;
 q_final:=modelo_inverso2(destino);  // Recibimos del script
…
```

7. Movimiento rectilíneo

Como se ha visto anteriormente, lo importante del movimiento articular es que cada una de las articulaciones realicen su movimiento con la misma duración hasta alcanzar la posición final. No se tenía en cuenta la trayectoria que iba a recorrer la herramienta durante el proceso.

En el caso que nos ocupa ahora lo importante es que se mantenga una trayectoria rectilínea; es decir, tanto la posición como la orientación deben evolucionar de forma lineal. La interpolación a realizar ha de ser cartesiana. En cada uno de los puntos intermedios que se tomen para calcular la trayectoria, se deberá proceder al cálculo del modelo inverso para conocer el valor que tomará cada una de las articulaciones para alcanzar dicha situación intermedia.

En cuanto a la evolución de la posición, se puede resolver de manera sencilla, puesto que al ser trayectorias cartesianas habrá que interpolar linealmente cada

una de las coordenadas. Así, por ejemplo, si se quisiera interpolar desde el punto (X_0,Y_0) hasta el punto (X_1,Y_1) a velocidad constante las coordenadas deberían evolucionar de la siguiente manera:

$$X(t) = X_0 + \frac{X_1 - X_0}{T} t$$

$$Y(t) = Y_0 + \frac{Y_1 - Y_0}{T} t$$

donde T es el tiempo de duración del movimiento, el cociente $(Y_1 - Y_0)/T$ representa la velocidad a la que debe evolucionar cada coordenada. y t simboliza el instante actual del movimiento.

En cuanto a la evolución de la orientación, el problema se vuelve más complejo, debido a que no se puede realizar una interpolación de la rotación al estar representada en una matriz de 3x3.

La utilización de un par de rotación resulta la manera más adecuada para generar la trayectoria cartesiana de orientación. La evolución más natural desde una orientación inicial a otra final será aquella que haga girar de manera progresiva el extremo del robot desde su orientación inicial hasta la final en torno a un eje de rotación fijo.

Dado un sistema ortonormal inicial y otro final rotado respecto del primero, existe un único eje **k** que permite pasar del sistema inicial al final girando un ángulo θ respecto a él. Por lo tanto, es necesario buscar cual es el par (**k, θ**) que relaciona los sistemas de coordenadas ortonormales asociados a ambas orientaciones, y realizar la evolución temporal mediante un giro en torno al eje **k** de valor.

$$\theta(t) = \frac{\theta_{Final}}{T} t$$

De esta manera, el ángulo evolucionará desde 0 hasta su valor final durante un tiempo T.

Para poder realizar esta solución se ha añadido al modulo algebra.pas un tipo de dato y algunas funciones:

```
Type
TPar_Rotacion=record
                r:TCoordenadas;
                tita:single;
              end;

function transformacion_a_par(T:TTransformacion):TPar_rotacion;

function par_a_transformacion(par:TPar_rotacion):TTransformacion;
```

Para alcanzar la solución, primero se debe calcular la transformación de destino relativa a la inicial. Esta trasformación se expresará mediante un par de rotación, la componente tita de dicho par de rotación será el ángulo que habrá que

interpolar, permaneciendo el eje constante. La implantación de esta solución se puede observar en el procedimiento *mover_linea_recta* de la clase *TRobot*.

```
Procedure TRobot.mover_linea_recta(destino:TTransformacion);
…
oriTdest:=multiplicar(invertir(origen),destino);
 p:=transformacion_a_par(oriTdest);
 Transformacion_inicial:=origen;
 Transformacion_intermedia:=origen;
 Transformacion_final:=destino;
...
end;
```

Hay que tener en cuenta que la trasformación intermedia obtenida será una transformación relativa a la transformación inicial. Dicha transformación habrá que convertirla en una transformación absoluta. Una vez realizado, se procederá al cálculo del modelo inverso para obtener las coordenadas articulares que generen dicha transformación.

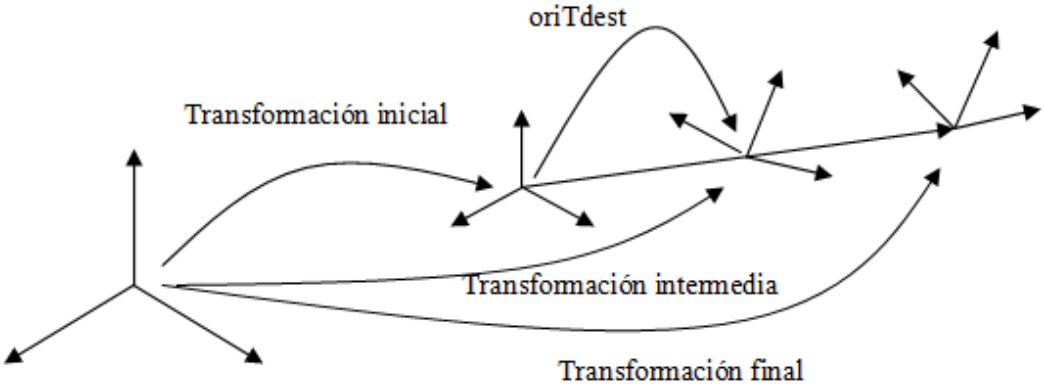


Figura 54. Relación de la transformación intermedia.

El procedimiento se ha implementado en **RobotScene** en el procedimiento *mover* de la clase *TRobot*.

```
procedure TRobot.mover(tiempo:double);
...
if estado=movimiento_rectilineo then
 begin
   t_actual:=t_actual+tiempo;
   if t_actual<T then
     begin
     x:=transformacion_intermedia.pos.x+velocidades2.x*tiempo;
     y:=transformacion_intermedia.pos.y+velocidades2.y*tiempo;
     z:=transformacion_intermedia.pos.z+velocidades2.z*tiempo;
     p.tita:=p.tita+velocidades2.tita*tiempo;
     transformacion_intermedia:=multiplicar(transformacion_inicial,
                                                  par_a_transformacion(p));
     transformacion_intermedia.pos.x:=x;
```

```
    transformacion_intermedia.pos.y:=y;
    transformacion_intermedia.pos.z:=z;
   transformacion_intermedia:=Definir_transformacion(x,y,z,par_rotacion);
    q_intermedio:=modelo_inverso2(transformacion_intermedia);
      for i:=1 to numero_articulaciones do
        begin
         articulacion:=articulaciones[i] as TArticulacion;
         articulacion.valor:=q_intermedio[i];
        end;
end ;
```

De igual manera que para el apartado anterior la variable destino debe ser la relativa al robot y no la absoluta.

8. Movimientos relativos

En RobotScene existe la posibilidad de especificar el destino relativo a la transformación de la herramienta, o si el robot no tuviera añadida una herramienta, el destino podría ser relativo a la transformación del último de los sólidos del robot.

Para operar con el destino relativo se debe transformar en absoluta la transformación relativa que nos pasan como parámetro.

```
procedure TRobot.mover_relativo(destino:TTransformacion);
...
modelo_directo(x,y,z,roll,pitch,yaw);
Tr:=Definir_transformacion(x,y,z,roll,pitch,yaw);
destino:=multiplicar(Tr,destino);
...
```

A partir de este punto se procede de la misma manera que en apartados anteriores, transformando *destino* en una transformación relativa al robot y llamando al modelo inverso.

9. Gestión de la herramienta.

La gestión de la herramienta y su explotación se realiza desde la clase TRobot, donde se encuentran todas las funciones necesarias para el manejo de la herramienta.

Para que los robots puedan trabajar con herramientas se debe añadir una herramienta al robot. Asociar una herramienta a un robot se realiza mediante el procedimiento *colocar_herramienta*:

```
procedure TRobot.colocar_herramienta(ruta:string);
```

Este procedimiento recibe como parámetro la ruta del archivo de herramienta *.herr* de la herramienta que queramos añadirle. Utilizando dicha instrucción se creará en memoria una herramienta que se asocia a su correspondiente robot por medio del campo *herramienta* de la clase *TRobot.* El campo en el cual se guarda la herramienta es del tipo *THerramienta*. En este punto el programa realizará las

transformaciones necesarias para realizar el cambio de referencia asociado a la herramienta.

Las herramientas que sean del tipo garra pueden interactuar con los demás objetos del escenario.

Para poder operar con la herramienta la clase *TRobot* incorpora los procedimientos *abrir* y *cerrar* que realizan llamadas a la clase THerramienta en la que se describe el movimiento de la herramienta.

```
procedure TRobot.abrir;
procedure TRobot.cerrar(cuanto:single);
```

Al utilizar cualquiera de estos procedimientos el estado del robot se actualiza a *accionando_herramienta.* Debido al cambio de estado, se ejecutará por medio del *cadencer* el procedimiento *mover* de la clase *TRobot* que, a su vez realiza una llamada a la clase THerramienta donde se ejecuta realmente el control de movimiento de la herramienta.

```
procedure TRobot.mover(tiempo:double);
...
if estado=accionando_herramienta then
 begin
  herramienta.mover(tiempo);
  if herramienta.estado=parada then
    begin
       estado:=parado;
       recien_parado:=true;
    end;
 end
...
```

Como se puede observar en el código, este procedimiento evalúa el estado de la herramienta de forma que el robot cambiará de estado si así lo ha hecho su correspondiente herramienta.

Desde este procedimiento se llama a la función *mover* de la clase *THerramienta* que realizará los cálculos necesarios para el movimiento de la garra. De dicho procedimiento hablaremos más adelante cuando se explique que operaciones se realizan cuando la herramienta agarra los objetos del escenario.

10. Agarrar y soltar objetos.

Como se ha podido comprobar en los apartados anteriores, el *cadencer* ejecuta las rutinas necesarias para la realización de los movimientos robóticos y, a continuación, muestra por pantalla el movimiento del robot. Cuando se está calculando el movimiento del robot, la clase *TRobot* llama a la función mover de la clase *THerramienta*, que realizará tanto los cálculos necesarios para el movimiento de la garra, como las operaciones oportunas para detectar colisiones entre la garra y los objetos del escenario.

Con el fin de manipular los objetos del escenario es imprescindible la inclusión de un detector de colisiones. Para la detección y supervisión de colisiones GLScene cuenta con un componente denominado TCollisionManager que permite detectar colisiones de forma automática y ejecutar código ante esa detección.

Lla comprobación de colisiones se realiza de manera sencilla con este componente; la detección tiene lugar cada vez que realizamos la llamada al procedimiento *CheckCollisions* del componente *TCollisionManager*.

```
procedure TCollisionManager.CheckCollisions
```

Este procedimiento detectará las colisiones y generará un evento por cada colisión que haya sido encontrada. El evento generado se denomina *OnCollision* y forma parte del componente *TCollisionManager*. Es por tanto en dicho evento donde se deben escribir las instrucciones que describan el comportamiento del programa cuando se produzcan las colisiones.

Como el detector de colisiones forma parte de la aplicación principal será necesario la utilización de flags que indiquen el estado del robot fuera de la clase *TRobot.*

Resumiendo, la supervisión se llevará a cabo en el procedimiento *mover* de la clase *THerramienta* que habrá sido disparado previamente por el correspondiente procedimiento *mover* de la clase *TRobot;* este a su vez se habrá ejecutado por un disparo del *cadencer.* En la siguiente muestra de código se puede observar como se calculan los movimientos de la garra y como se realiza la llamada al detector de colisiones.

```
procedure THerramienta.mover(tiempo:double);
begin
  if estado=abriendo then
  begin
    apertura:=apertura+tiempo*velocidad;
    if apertura=apertura_maxima then
    estado:=parada;
  end else  // estamos cerrando
  if estado=cerrando then
  begin
     apertura:=apertura-tiempo*velocidad;
     if apertura=apertura_final then
     begin
        estado:=parada;
        agarrar:=true;
        if agarrar then
        begin
          Ventana_escenario.supervisor_colisiones.CheckCollisions;
          agarrar:=false;
        end;
     end;
  end;  // estamos cerrando
```

```
end;
```

Una vez realizada la llamada al detector de colisiones se generará el evento *OnCollision* que provocará la ejecución del código diseñado para tratar las colisiones. Si se cumplen una serie de requisitos la garra atrapara el objeto, y estará en condiciones de manipular el escenario, por ejemplo, trasladando el objeto agarrado a otro punto de la escena.

```
procedure TVentana_escenario.supervisor_colisionesCollision(Sender: TObject;
object1,  object2: TGLBaseSceneObject);
var solido1,solido2:TSolido;
   n1,n2:string;
   robot:TRobot;
begin
  solido1:=object1.parent as TSolido;
  solido2:=object2.parent as TSolido;
  if solido1<>solido2 then
  begin
    if ((solido1.tipo=de_escenario) and (solido2.tipo=de_herramienta)) or
      ((solido2.tipo=de_escenario) and (solido1.tipo=de_herramienta)) then
      // Un robot ha cogido algo
      begin
        if solido1.tipo=de_herramienta then
        begin
          robot:=((solido1.Parent as THerramienta).robot as TRobot);
          if robot.Herramienta.agarrar then
          begin
            robot.coger(solido2.Parent as TObjeto_escena);
          end;
        end else
        begin
          robot:=((solido2.Parent as THerramienta).robot as TRobot);
          if robot.Herramienta.agarrar=true then
          begin
            robot.coger(solido1.Parent as TObjeto_escena);
          end;
       end;
      end;
  end;
end;
```

Como se puede ver en la cabecera del procedimiento que maneja el evento, existen dos parámetros, object1 y object2, donde se guardan los sólidos que han producido dicha colisión. En el código se examina si la colisión ha sido entre una herramienta y un objeto del escenario. Si se cumple la condición el robot agarrará el objeto mediante el procedimiento *coger* de la clase *TRobot*.

```
   procedure TRobot.coger(objeto:TObjeto_escena);
```

El procedimiento *coger* añadirá el sólido que produjo la colisión con la herramienta a la correspondiente cadena cinemática del robot. Adicionalmente, el procedimiento crea un enlace virtual entre el sólido y el robot de manera que a partir de ese momento realizarán el movimiento de forma conjunta, como si el sólido agarrado fuera una parte más de la herramienta

Si por el contrario, se le transmite al robot la orden de que abra la garra mediante instrucciones del tipo *soltar*, el estado del robot cambiará, y por tanto, se producirá una llamada al método *mover* de la clase *THerramienta* que calculará las nuevas posiciones de los sólidos que componen la herramienta. Posteriormente, se ejecutará el método *dejar* de la clase *TRobot*, que eliminará el vínculo virtual creado entre el robot y el objeto atrapado por la garra.

```
procedure TRobot.dejar;
```

Por último, es conveniente recordar que las librerías GLScene no tratan adecuadamente la colisión con objetos de formas complejas creados mediante extrusión, revolución, o por el método tuberías. Para corregir estos problemas se implementaron, en la versión mono-robot de RobotScene, dos procedimiento denominados *corregir_colisiones* uno de ellos se encuentra en la clase *TSolido* y el otro en la clase *TRobot*, que deben ejecutarse al añadir un objeto o un robot la escena, respectivamente.

11. Capacidad multi-robot. Creación y gestión de una ventana de programación por robot

Para dotar a **RobotScene** con capacidad multi-robot se han rediseñado ciertas cuestiones. En este punto se analizarán las primeras medidas adoptadas para capacitar al simulador de uso y explotación de varios robots.

Inicialmente **RobotScene** estaba diseñado con la idea de utilizar un único robot en el escenario. Bajo esa premisa, se crea en el programa una variable global que guardará la información relativa al último robot seleccionado de la escena, dicha variable se denomina *Robot_activo*. La variable es utilizada en todo momento que se hace referencia al robot que gobierna la escena.

Conocida la existencia de esta variable el primer paso para conseguir la capacidad multi-robot es la creación de un vector de robots que nos permita guardar los datos de hasta 3 robots.

```
TVector_robots = Array[1..Max_robots] of TRobot;
```

Creado el vector de robots, se modifican las instrucciones relativas a la carga del escenario en ficheros, de forma que se almacenen en el vector los robots de la escena.

```
procedure cargar_escenario(archivo:string;arbol:TTreeView; escena:TGLDummyCube;
var programa:string;  var vector_robotsactivos:TVector_robots;
var  numero_robots:integer);
…
```

```
if tipo='Robot' then
  begin
  …
  vector_robotsactivos[numero_robots]:=robot;
  …
```

Para que cada robot funcione independientemente en el escenario fue necesario crear una ventana de programación por robot. Para guardar dichas ventanas de programación se creó un vector de ventanas.

```
  Vector_scripts: Array [1..3] of TVentana_programacion;
```

La creación de las ventanas de programación se llevó a cabo creando varias instancias de la clase *TVentana_programación*. La creación de una ventana de programación se produce cuando se crea o se carga un nuevo robot en la escena. En el siguiente código puede apreciarse como se realiza la instanciación de las ventanas.

```
// Creo su correspondiente ventana de programación
Application.CreateForm(TVentana_programacion, Vector_scripts[numero_robots]);
```

Cuando ya se ha conseguido la creación de una ventana de programación por cada robot es el momento de gestionar la carga y guardado de cada script de programación. Fue necesario, nuevamente, la modificación del módulo *ficheros.pas*, de forma que se crearan dentro de la carga de ficheros las ventanas de programación.

```
procedure cargar_escenario(archivo:string;arbol:TTreeView; escena:TGLDummyCube;
var programa:string;  var vector_robotsactivos:TVector_robots;
var  numero_robots:integer);
...
  // Cargamos los script de programación
  Application.CreateForm(TVentana_programacion, Vector_scripts[numero_robots]);
...
```

Por lo tanto, la estructura que guardaba los datos en formato *.esc* tuvo que ser convenientemente actualizada. Asímismo, se realizaron los cambios convenientes para permitir la carga de escenarios de la versión mono-robot del programa, que requiere la evaluación previa de la versión del fichero *.esc* que se va a cargar.

```
procedure cargar_escenario(archivo:string;arbol:TTreeView; escena:TGLDummyCube;
var programa:string;  var vector_robotsactivos:TVector_robots;
var  numero_robots:integer);
...
  if (s='Programa:') then
  begin
    version:=0;  // RobotScene "monorobot"
  end
  else begin
```

```
    version:=1;  // RobotScene "multirobot"
  end;
...
```

Cuando se produce la eliminación de un robot o la eliminación completa del escenario, cada instancia de la ventana de programación debe ser convenientemente cerrada y liberada de la memoria.

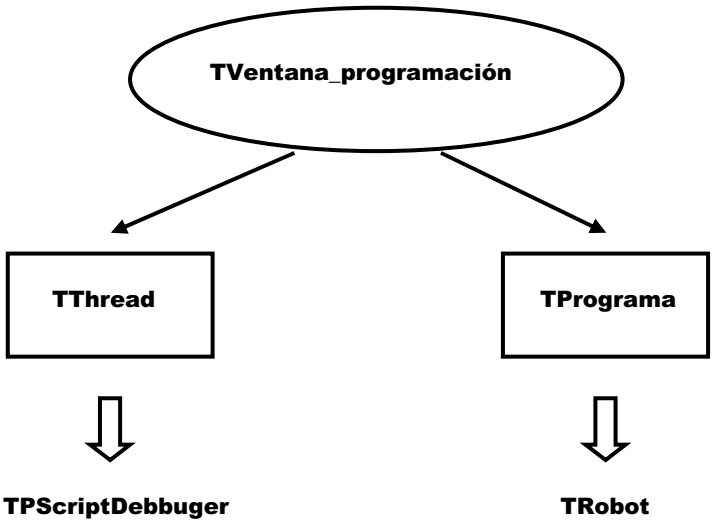
```
procedure TVentana_escenario.boton_eliminarClick(Sender: TObject);
...
if nodo.tipo=nodo_robot then
begin
    robot:=nodo.objeto as TRobot;
    Vector_scripts[robot.lugar].Accion_resetExecute(Self);
    Vector_scripts[robot.lugar].Close;
    Vector_scripts[robot.lugar].Free;
    ...
...
```

El programa está en este momento capacitado para crear y gestionar ventanas de programación para varios robots. De este modo, el programa ha sido rediseñado para realizar las mismas operaciones sobre un robot pero en este caso se maneja la información por medio de vectores de modo que toda ella se almacena en estructuras que serán útiles cuando se quiera aplicar movimiento para varios robots.

12. Capacidad multi-robot. Concurrencia

Con todos los cambios descritos en el apartado anterior el programa ya está preparado para la utilización de varios robots; sin embargo, queda un aspecto muy importante que resolver. El programa debe ser capaz de ejecutar varios programas robóticos simultáneamente. Para esta tarea se utiliza la clase *TThread* que proporciona Delphi y permite la creación de varios hilos de ejecución.

Cuando se van a utilizar varios hilos de programación hay que pensar qué componentes y variables utilizadas en el programa deben estar contenidos dentro de la clase *TThread.* Por tanto, se han vuelto a modificar las relaciones entre las diferentes clases de variables quedando la estructura de la ventana de programación de la siguiente manera:



Como se puede ver en el esquema, la estructura de variables en la clase *TVentana_programacion* ha sido rediseñada de manera que la ventana de script contendrá dos variables. Por un lado, una variable del tipo *THilo*, que a su vez es descendiente del tipo *TThread*, que estará compuesta por un *debugger* que ejecutará un script de programación. Por otro lado, una variable de tipo *TPrograma* que ejecutará sus instrucciones robóticas sobre una de sus variables de tipo *TRobot.*

Todas las instrucciones relativas a los diferentes hilos de ejecución se encuentran en el módulo *script_programa.pas.* En dicho módulo se puede observar como actúan los hilos de ejecución.

Se crea un hilo de ejecución cuando se inicia la ejecución de la tarea robótica por medio de los controles de la ventana de programación. En el siguiente código pueden observarse las acciones que realiza la aplicación cuando se pulsa el botón que inicia la ejecución del script:

```
procedure TVentana_programacion.Accion_EjecutarExecute(Sender: TObject);
begin
  if Hilo=nil then  // Si no existe el hilo
  begin
    Hilo:=Thilo.Create(programa.Robot.Lugar);  // Creo el hilo
  end else  // Si existe el hilo
  begin
    Hilo.Pausa:=false;
    if Hilo.Suspended=true then  // Si el hilo esta suspendido
    begin
      Hilo.Resume;  // Continuo la ejecución del hilo
      Mensajes.Items.Add('Ejecutando...');
      Mensajes.TopIndex:= Mensajes.Items.Count-1;
      Mensajes.Refresh;
    end;
  end;
```

```
  // Se activan flags de estado
  ejecutando_programa:=true;
  programa_activo:=true;
end;
```

Cuando se produce la creación de un hilo de ejecución, el nuevo hilo ejecutará automáticamente las instrucciones contenidas en el método *execute* de la clase *THilo*. En el siguiente código se puede observar el conjunto de instrucciones de dicho procedimiento.

```
//////////////////// Código de ejecución del hilo  //////////////
procedure THilo.Execute;
begin
  Ejecutar_instruccion;
  Terminate;
end;
```

Este procedimiento realiza una llamada a *ejecutar_instruccion.* Este nuevo procedimiento inicia la ejecución del script mediante una variable del tipo *TPSScriptDebugger.* Una vez que es el script es puesto en ejecución se desencadena un proceso repetitivo que finaliza con la ejecución de la última instrucción del programa robótico. Este proceso repetitivo se produce por el disparo continuo de un evento en los tiempos en espera del script.

El evento se denomina *OnDebuggerIdle* y pertenece a la clase *TPSScriptDebugger*. Mediante la utilización de este evento podemos controlar la ejecución del programa robótico,y adicionalmente, se pueden crear los controles manuales, para el tipo de ejecución del programa, de forma relativamente sencilla. En el modo normal de ejecución, la transición de una instrucción a la siguiente se realiza de forma automática. En cambio, en el modo paso a paso la transición se realiza de forma manual controlada por el usuario.

La siguiente pieza de código muestra, mediante la utilización de este evento, la suspensión de la ejecución del script hasta que el robot haya terminado la ejecución de la última instrucción; es decir, hasta que el estado del robot sea *parado*. Como las instrucciones de movimiento duran un tiempo determinado, se suspende la ejecución del script hasta que el robot este completamente parado. Como el disparo del evento se produce dentro del nuevo hilo de ejecución, las acciones que realiza dicho evento se ejecutan en el nuevo hilo de ejecución.

```
procedure THilo.DebuggerIdle(Sender: TObject);
begin
  Application.ProcessMessages;
  if ((Debugger.Exec.Status in isRunningOrPaused) and
(Vector_scripts[posicion].ejecutando_programa)) then
  begin
    if Vector_scripts[posicion].programa.Robot.estado=parado then
    begin
      ejecutar_instruccion;
```

```
      Sleep(20);
    end;
```

Tras la finalización de todas las instrucciones del script de programa, el hilo se liberara automáticamente de la memoria. La utilización del procedimiento nos permite indicar al hilo que acciones acometer en el momento de su destrucción.

```
////////////////////// Destrucción del hilo  //////////////////////
destructor THilo.Destroy;
begin
  Debugger.Free;
  Vector_scripts[posicion].Hilo:=nil;
  inherited;
end;
/////////////////////////////////////////////////////////////////////////////
```

La representación por pantalla de los movimientos del robot se realiza a través de un cadencer que ejecutará un procedimiento repetitivo. Para la versión multi-robot ha sido necesario modificar dicho procedimiento para que se llame a la instrucción mover de la clase TRobot por cada robot en el escenario. Esta explicación puede observarse en el siguiente código.

```
procedure TVentana_escenario.GLCadencer1Progress(Sender: TObject; const deltaTime,
            newTime: Double);
…
Begin
…
for i:= 1 to numero_robots do
    begin
      if Vector_scripts[i].programa.Robot<>NIL then
      begin
        Vector_scripts[i].programa.Robot.mover(deltatime);
...
```

Con todas estas modificaciones, se ha dotado al simulador con la capacidad para ejecutar cada script en un nuevo hilo de ejecución. Sin embargo, el uso de hilos obliga al programador a tomar ciertas precauciones. Cuando se ejecutan varios hilos de forma concurrente se debe evitar que los hilos accedan a partes de memoria de manera simultánea. En algunos casos, el programador estará obligado a sincronizar instrucciones. Para estos menesteres la clase *TThread* incorpora el método *Synchronize*. Este método permite seleccionar instrucciones que serán ejecutadas por el hilo principal de la aplicación.

Este será el modo en el que se ejecutarán instrucciones en el simulador; bien sea porque la ejecución de ese tipo de instrucciones pueda conllevar algún riesgo, o bien, para sincronizar los gráficos con la VCL. Los gráficos deben ser ejecutados por el hilo principal, si no se hace así, es posible que en algunos casos los gráficos no se visualicen de la forma correcta.

En **RobotScene** se ha utilizado el método *Synchronize* en algunos casos. En el siguiente extracto de código se puede observar la utilización del método *Synchronize* en el método *ejecutar_instruccion* de la clase *THilo*.

```
procedure THilo.ejecutar_instruccion;
begin
  …
  if Vector_scripts[posicion].grabando then
  begin
     Synchronize(Ventana_escenario.grabar);
  end else
  …
end;
```

En la clase *TRobot* ha sido obligatorio sincronizar las instrucciones de aviso de errores; ésto ha sido debido a que los avisos se muestran bajo una ventana gráfica que debe ser sincronizada con el hilo principal. Con tal propósito ha sido creada una variable de tipo descendiente de la clase *TThread*.

A continuación se muestra el método *execute* de esta nueva clase descendiente de *TThread*. En el código se puede observar que se sincroniza el mensaje de aviso mediante otro procedimiento denominado *mostrar_mensaje*.

```
procedure THilo_mensaje.Execute;
begin
  Synchronize (Mostrar_mensaje);
  Terminate;
end;
```

```
procedure THilo_mensaje.Mostrar_mensaje;
begin
  Showmessage(mensaje);
end;
```

13. Ejecución de un programa robótico, la clase TPrograma.

La clase *TPrograma* actúa como contenedor de todas las funciones y procedimientos que componen el lenguaje robótico. Más adelante se estudiará la solución implementada en el simulador para llamar a estas instrucciones desde el script.

En esta clase se definen todos los procedimientos que posteriormente se asociarán al script. En algunos casos, los procedimientos que ejecuta el script están definidos en otras unidades del programa. Por ejemplo, en la clase *TRobot* se definen los movimientos del robot, en la unidad *algebra.pas* se define el tratamiento con las matrices. En otros casos, los procedimientos serán completamente nuevos y estarán definidos en la clase *TPrograma*. De cualquier manera, esta clase tiene el objetivo de unificar todos los procedimientos que se ejecutarán a través del script.

Dentro de la clase *TPrograma* se encuentra una variable del tipo *TRobot*. Esta variable tiene como objetivo la asociación de un programa con un robot concreto, de esta manera, cada script ejecutará los procedimientos únicamente sobre su correspondiente robot asociado.

```
TPrograma=class
    Robot:TRobot;
    arbol:TTreeview;
    archivo:string;
    …
end;
```

En esta pieza de código se ha podido observar la declaración de la clase programa, y como se declara una variable del tipo *TRobot* que será posteriormente utilizada para asociar cada script con un robot.

Las funciones y los procedimientos contenidos en la clase *TPrograma* son las instrucciones del lenguaje robótico que serán detalladas en el apartado correspondiente.

14. Tratamiento del script del programa.

Como se ha comentado anteriormente se ha utilizado el componente *TPSSCriptDebugger* para realizar el motor de scripts y ha sido correctamente encapsulado dentro de la clase *THilo*.

Este componente permite importar procedimientos del código fuente, que posteriormente pertenecerán al lenguaje robótico. Estos procedimientos pasarán a ser conocidos por el compilador del programa y el usuario del simulador podrá utilizarlos para programar su tarea robótica.

Para indicar al script qué procedimientos es capaz de importar, lo haremos mediante la función *AddMethod*, de la misma manera que se realiza en RobotScene a la hora de tratar la cinemática inversa.

En el evento producido al compilar el programa robótico, generado por el motor de scripts, se añaden todas las primitivas de las que consta el lenguaje robótico. Los tipos de datos que se añaden al lenguaje robótico mediante el procedimiento *AddTypeS*:

```
  // Tipos que soporta el script
Sender.Comp.AddTypeS('TCoordenadas','record x,y,z:single; end;'). ExportName:=true;
Sender.Comp.AddTypeS('TTransformacion','record pos,n,o,a:TCoordenadas; end;').
ExportName:=true;
...
// Primitivas del lenguaje robótico
  Sender.AddMethod(Self, @THilo.MovCoor,
     'procedure MovCoor(destination:TTransformacion)');
  Sender.AddMethod(Self, @THilo.MovLin,
     'procedure MovLin(destino:TTransformacion)');
```

```
...
```

De esta forma, todos los procedimientos y funciones que se añadan podrán usarse dentro del script. Tal como muestra el código, algunas instrucciones no realizan una llamada directa a la clase *Tprograma*. Si se realiza una llamada directa a la clase *Tprograma* no es posible indicar al programa qué robot usar; en este caso, el script ejecuta la función desde una instancia de la clase *Tprograma* que no tiene asociada ningún robot. Para indicar qué variable del tipo *Tprograma* es utilizada, y por tanto que robot tiene asociado, se han implementado unas funciones, en la clase *THilo,* que actúan como lanzadores de las instrucciones.

En el siguiente código puede observarse como funciona uno de estos lanzadores. El hilo realiza la llamada a los procedimientos a través de la variable *programa* contenida en la clase *TVentana_programacion.*

```
procedure THilo.MovLin(destination:TTransformacion);
begin
  vector_scripts[posicion].programa.MovLin(destination);
end;
```

Para remarcar las palabras clave en el editor de programas se utiliza el componente *TSynEdit*. Mediante el evento *OnSpecialLineColors* se permite al editor remarcar la línea de código que se esta ejecutando en ese instante, y también, señalar las líneas de error en la compilación del programa.

15. Capacidad multi-robot. Comunicación entre robots

Para realizar la comunicación entre robots se ha realizado un motor de gráficos en dos dimensiones basado en GLScene. La parte gráfica permite la creación de cables mediante eventos del tipo *drag and drop*, es decir, los cables se crean arrastrando con el ratón desde un *pad* de salida a uno de entrada.

Para implementar las conexiones cableadas en **RobotScene** se han utilizado matrices. Estas matrices se modifican cuando se añade o elimina algún cable.

Como puede observarse en el código, se ha implementado una matriz por robot de diez filas por veinte columnas, de manera que el robot pueda conocer el estado de su cableado con otros robots mediante una única matriz. Como el escenario esta diseñado para albergar como máximo a tres robots, se ha creado un vector de matrices que encapsula las matrices de cada robot.

```
TMatrizCableado = Array [0..9] of Array [0..19] of boolean;
TVectorMatrices = Array [1..3] of TMatrizCableado;
```

Cuando se produce la creación de un cable, se rellena automáticamente su correspondiente matriz,

```
procedure TVentana_Cableado.GLSceneViewer1EndDrag(Sender, Target:
TObject; X, Y: Integer);
...
```

```
Begin
…
VectorMatrices[VCable[Numero_Cables].Robot_Salida][VCable[Numero_Cables
].Salida][k+VCable[Numero_Cables].Entrada]:=true;
…
End;
```

El borrado de datos de la matriz, se realiza cuando se elimina un cable o también cuando se elimina algún robot. Para la eliminar datos de la matriz se han implementado en **RobotScene** dos procedimientos. El primero se ejecuta cuando se elimina un cable. El segundo se ejecuta cuando se elimina un robot.

```
procedure TVentana_Cableado.BorrarDatoMatriz(NumeroCable: Integer);
var
  k: Integer;
begin
/////////////// Borramos dato de la matriz de cableado ///////////////
…
VectorMatrices[VCable[NumeroCable].Robot_Salida][VCable[NumeroCable].Sal
ida][k+VCable[NumeroCable].Entrada]:=false;
  ////////////////////////////////////////////////////////////////////
end;
```

```
// Este método adecua la matriz al eliminar un robot
procedure TVentana_Cableado.RefrescarMatriz(RobotEliminado: integer;
NRobotsAntesDeEliminar: integer);
```

Cada cable que se cree en el editor, simula un cableado real entre la salida de un robot y la entrada de otro. Es decir, que cuando la salida cambie su estado la entrada cableada del otro robot deberá cambiar su valor automáticamente. Para activar y desactivar la entrada de un robot cuando se activa o desactiva la salida cableada de otro robot, se han creado dos procedimientos.

```
procedure ActivarEntradasConectadas(nRobot,nSalida:integer);
procedure DesactivarEntradasConectadas(nRobot,nSalida:integer);
```

En la siguiente pieza de código puede observarse cómo el robot emplea estas funciones cuando cambia el estado de sus salidas mediante la función *signal* de la clase *TRobot*,

```
procedure TRobot.signal(i:integer);
begin
  if (i>0) and (i<=10) then
  begin
    salidas[i]:=true;
    Ventana_Cableado.ActivarEntradasConectadas(Lugar,i);
  end else
  if (i<0) and (i>=-10) then
  begin
```

```
    salidas[i*sign(i)]:=false;
    Ventana_Cableado.DesactivarEntradasConectadas(Lugar,i*sign(i));
  end else
  ...
```

Para permitir guardar los archivos de cableado en formato *.cbl,* se ha creado un procedimiento denominado *guardar_cableado*. Este procedimiento crea un fichero de texto que contiene los datos relativos a todos los cables del escenario que han sido previamente almacenados en un vector de cables.

```
procedure guardar_cableado(archivo:string; VectorCables: TVectorCable);
```

Cuando se necesita cargar un cableado previamente guardado, se ha implementado otro procedimiento.

```
procedure cargar_cableado(archivo:string; var VectorCables: TVectorCable);
```

Por último, ha sido necesario modificar en la unidad *ficheros.pas* los procedimientos que permiten la carga y guardado del escenario, para almacenar dentro de los ficheros de escenario, la ruta del fichero de cableado. De esta manera se puede cargar automáticamente el fichero de cableado cuando se carga un escenario.

En la siguiente porción de código se observan las modificaciones en el procedimiento *guardar_escenario* en la unidad *ficheros.pas*.

```
guardar_escenario(archivo:string;arbol:TTreeView;Cableado:string);
...
begin
  Assignfile(f,archivo);
  Rewrite(f);
  Write(f,'Cableado: ');
  if Cableado='' then WriteLn(f,'No hay.')
                 else WriteLn(f,ExtractRelativePath(archivo,Cableado));
  ...
end;
```

Finalmente, el siguiente trozo de código muestra las modificaciones en el procedimiento *cargar_escenario* en la unidad *ficheros.pas*.

```
procedure
cargar_escenario(archivo:string;arbol:TTreeView;escena:TGLDummyCube;
...
Begin
...
    Leer_cadena; // 'Cableado: '
    readln(f,s);
    if s='No hay.' then
    begin
      RutaCableado:=''
```

```
    end
    else begin
      RutaCableado:=extractfilepath(archivo)+s;
    end;
...
```

6.5 DESCRIPCIÓN DEL LENGUAJE DE PROGRAMACIÓN DE ROBOTS

Tal como se anunció en el apartado "Análisis de soluciones" el desarrollo de la versión multi-robot de **RobotScene** supone la elaboración de procedimientos que permitan el desarrollo de tres tareas que la versión mono-robot no permitía llevar a cabo: creación y gestión de ventanas de programación para cada robot, ejecución simultánea de instrucciones de programa y establecimiento de comunicación efectiva entre los robots.

En lo que sigue, se explica qué instrucciones conforman el lenguaje robótico. Dichas instrucciones robóticas son las mismas para una versión mono-robot que para una versión multi-robot, pues estas instrucciones se ejecutan sobre un único robot, y son generalmente instrucciones de movimiento. Además, la versión mono-robot de **RobotScene** ya contaba con instrucciones para manejar las entradas y salidas binarias. Dichas instrucciones han sido modificadas en el código fuente del programa para implementar una solución multi-robótica; sin embargo, para el usuario, las instrucciones tienen el mismo nombre y funcionan de la misma manera. La única modificación realizada en estas instrucciones respecto a la versión mono-robot ha sido el cambio de nombre de la antigua instrucción *stop* por *pause*.

Como consecuencia de lo comentado en el párrafo anterior, el lenguaje robótico utilizado se ajusta a lo que se recoge a continuación que es, en realidad, copia del lenguaje empleado en la versión mono-robot.

1. Tipos de datos estándares.

El lenguaje de programación habilitado para la programación de las aplicaciones robóticas es un lenguaje basado en Pascal y tiene su misma sintaxis. Por esta razón, se podrán utilizar gran parte de los datos definidos para el lenguaje Pascal estándar.

Tipos enteros

Tipo	Rango	Formato
Integer	–2147483648 a 2147483647	Con signo 32 bits.
Cardinal	0 a 4294967295	Sin signo 32 bits.
Shortint	–128 a 127	Con signo 8 bits
Smallint	–32768 a 32767	Con signo 16 bits.
Longint	–2147483648 a 2147483647	Con signo 32 bits.
Byte	0 a 255	Sin signo 8 bits
Word	0 a 65535	Sin signo 16 bits
Longword	0 a 4294967295	Sin signo 32 bits.

Existen dos tipos base de 32 bits que son *integer* y *cardinal*. Las operaciones aritméticas entre tipos enteros, por regla general, devolverán un valor de tipo *integer*. A todos los efectos resultan equivalentes los tipos *Integer* y *Longint*, así como *Cardinal* y *Longword*. No hay problemas de asignación de diferentes tipos mientras el valor esté dentro del rango adecuado.

Tipos reales:

Tipo	Rango	Cifras	Tamaño
Single	1.5×10^{-45} a 3.4×10^{38}	7-8	4 bytes
Double	5.0×10^{-324} a 1.7×10^{308}	15–16	8 bytes
Extended	3.6×10^{-4951} a 1.1×10^{4932}	19–20	10 bytes
Currency	–922337203685477.5808 a 922337203685477.5807	19–20	8 bytes

Todos estos datos tienen una notación de coma flotante, excepto el tipo de dato *currency*, que se representa con coma fija. La representación de éste último se realiza internamente como un entero de 64 bits con los últimos 4 dígitos, los menos significativos, representando las cifras decimales.

Se recomienda insistentemente el uso del tipo de dato single, de 4 bytes (32 bits), puesto que muchas rutinas algebraicas están optimizadas para este tipo de dato, con lo que la ejecución resultará mucho más fluida.

Tipos carácter (char):

El tipo *char* puede contener un solo carácter. Cada uno de estos caracteres puede ser expresado gracias al código ASCII. Un carácter *char* se representa entre comillas simples, por ejemplo: 'A' 'b' '*' ' ' '5'.

Los tipos de datos carácter son ordinales, lo cual significa que cada carácter representa una posición en una serie ordenada (de 0 a 255). El orden de la letra A es 65, el de la letra B es 66, etc. Para saber el orden basta con mirar una tabla ASCII.

La función predefinida *chr* permite referenciar todos los caracteres, así, *chr*(65) equivale a la letra A, *chr*(66) equivale a la letra B, y así sucesivamente.

Tipo cadena (string)

Un tipo *string* (cadena) es una secuencia de caracteres correspondientes al código ASCII, de longitud indefinida, escrito en una línea sobre el programa y representado entre comillas simples, por ejemplo: 'cadena', 'otra cadena'. Se permite la cadena de longitud nula y se representa como ' '.

Tipo booleano

Una variable de tipo *booleano* puede tomar dos valores *true* o *false* (verdadero o falso), y surge como resultado de aplicar los operadores de comparación: =, >, <, >=, <= y <>.

Las operaciones que pueden realizarse con este tipo de datos son **not**, **and**, **or**, **xor**.

Tipo array (vector)

Este tipo de dato complejo almacena un conjunto de elementos del mismo tipo. Sus elementos son accesibles de forma independiente mediante un índice. El tamaño está fijado y deberá especificarse en el momento de declarar una variable de tipo array.

A continuación se muestra un ejemplo de la declaración de un tipo de variable vector:

```
Type nombre_array=array [1..8]  of single;
```

Es un vector de 8 elementos, con índices de 1 a 8. Para acceder a uno de sus elementos nos referiremos al índice, así, el segundo elemento será nombre_array[2] y será un tipo de dato single.

En el lenguaje robótico esta permitido la declaración y uso de *arrays* de varias dimensiones o matrices, accesibles desde dos índices. Ejemplo array[1..numero_filas,1..numero_columnas] of single.

Tipo registro (record)

Es un tipo de datos estructurado que almacena un conjunto de elementos que pueden ser del mismo tipo o de tipos diferentes.

Los componentes de un registro se denominan campos. Cada campo tiene un nombre, llamado identificador de campo, que es un identificador elegido por el programador cuando se declara el tipo de dato record. Un ejemplo de record se puede observar a continuación:

```
type

Mi_registro = record
                  Campo1:single;
                Campo2:integer;
              end;
```

Para acceder al primero de los campos se escribe: *Mi_registro.campo1*

2. Tipos de datos propios del lenguaje robótico:

Coordenadas de un punto:

Para referirnos a un vector tridimensional disponemos del dato *TCoordenadas*, cuya declaración es la siguiente:

```
type

TCoordenadas = record
                   x,y,z:single;
                 end;
```

Se puede observar en la declaración que el tipo *TCoordenadas* es de tipo *record*, y por esta razón se acceden a sus campos de la misma manera que en un *record*. Para referirnos a la componente x de un punto P (P es de tipo *TCoordenadas*) deberemos escribir *P.x*

Transformación homogénea:

Se ha definido el tipo de dato *TTransformación* que representa una matriz de transformación homogénea, que permitirá almacenar la posición y la orientación de un sistema de referencia:

```
type

TTransformacion=record
                    pos,n,o,a:TCoordenadas;
                end;
```

Para una transformación *T*, de tipo *TTransformación*, nos referiremos a la componente x de la posición mediante *T.pos.x*, o para referirnos a la componente n_y de la submatriz de rotación n, o, a mediante *n.y*.

3. Funciones algebraicas y trigonométricas:

El lenguaje robótico permite la utilización de una serie funciones algebraicas y trigonométricas para facilitar la programación de la tarea robótica:

```
function Sin(X: Single): Single;          // Seno
function Cos(X: Single): Single;          // Coseno
function Sqrt(X: Single): Single;         // Raíz
function Atan2(x,y:single):single;        // Arcotangente y cuadrante
function ACos(x:single):single;           // Arco Coseno
function degrees(rads:single):single      // Convierte a grados
function rads(degrees:single):single;     // Convierte a radianes
```

4. Instrucciones del lenguaje.

Existen tres tipos de primitivas dentro del lenguaje robótico. Las referentes a la importación de datos de la interfaz, las primitivas algebraicas para el trabajo matricial con transformaciones homogéneas y las primitivas de movimiento del robot. Las primitivas de movimiento del robot incluyen tanto las primitivas de uso de la herramienta como las primitivas para la gestión de las entradas y salidas binarias del robot.

Primitivas para la importación de datos.

Mediante una serie de funciones habilitadas, se puede incorporar a la tarea robótica transformaciones que habrán sido previamente definidas en la construcción del escenario. Se dispone de las dos funciones siguientes:

```
function  Import_Relative_Transform(s:string):TTransformacion;

function  Import_Absolute_Transform(s:string):TTransformacion;
```

Ambas funciones devolverán un valor del tipo TTransformación y toman como parámetro el nombre que tiene en el árbol el objeto que se quiere importar. La diferencia radica en que mediante *Import_Absolute_Transform* se importa la transformación absoluta del objeto, mientras que mediante la función *Import_Relative_Transform* se importa la transformación del objeto pero relativa a su antecesor en el árbol. Si se utiliza esta última técnica, para especificar la transformación total del objeto se deben realizar una serie de composiciones de transformaciones, recorriendo el árbol, mientras que si se importa de forma absoluta, toda esta serie de operaciones se realizará de forma interna y automática.

Primitivas para el trabajo con transformaciones homogéneas.

Estas primitivas permiten realizar operaciones básicas con el tipo de dato *TTransformacion*. Todas las rotaciones irán expresadas en grados y las distancias en milímetros.

function RotYPR(yaw,pitch,roll:single):TTransformacion;	Define una transformación a partir de los ángulos de roll, pitch y yaw.
function RotEuler(alfa,beta,gamma:single):TTransformacion;	Define una transformación a partir de los ángulos de Euler.
function RotX(angle:single):TTransformacion; **function** RotY(angle:single):TTransformacion; **function** RotZ(angle:single):TTransformacion;	Rotaciones en torno a los ejes principales.
function RotVector(axis:Tcoordenadas;angle:single):TTransformacion	Rotación en torno a un eje arbitrario. La especificación del eje de giro se hará mediante el parámetro de tipo Tcoordenadas.
function NullTr:TTransformacion;	Definición de la transformación nula, o lo que es lo mismo, la matriz identidad.
function Comp(T1,T2:TTransformacion):TTransformacion;	Composición (multiplicación) de dos transformaciones.
function Invert(T:TTransformacion):TTransformacion	Devuelve la inversa de una matriz.

function TransfYPR(x,y,z,yaw,pitch,roll:single):TTransformacion;	Define una transformación mediante x, y, z y los ángulos roll, pitch y yaw.
function TransfEuler(x,y,z,alfa,beta,gamma:single):TTransformacion;	Define una transformación mediante x, y, z y los ángulos alfa,beta,gamma.
function Translation(x,y,z:single):TTransformacion;	Define una traslación.
function Shift(T:TTransformacion; x,y,z:single):TTransformacion;	Traslada una transformación respecto de la referencia absoluta.
function ShiftRel(T:TTransformacion; x,y,z:single):TTransformacion;	Traslada una transformación respecto de la propia referencia.

Primitivas de uso del robot.

procedure procedure velocity(percent:single);	Define la velocidad de movimiento del robot en tanto por ciento de la máxima permisible.
procedure acceleration(percent:single);	Define la aceleración de movimiento del robot en tanto por ciento de la máxima permisible.
function where_tool:TTransformacion;	Nos devuelve la localización de la herramienta del robot, si la tuviera, en caso contrario nos devolverá la localización del último de los sólidos del robot.
function joints:TArray;	Devuelve el valor de las articulaciones del robot.
procedure set_configuration(conf:string);	Procedimiento para cambiar la configuración del robot.
procedure pause(ms:integer);	Detiene la ejecución de la aplicación durante un tiempo expresado en milisegundos.
procedure MovJoint(destination:Tarray);	Movimiento coordinado articular cuyo destino es un vector de coordenadas articulares que es tomado como parámetro.
procedure MovCoor(destination:TTransformacion);	Movimiento coordinado articular cuyo destino es una transformación.

procedure MovLin(destino:TTransformacion);	Movimiento rectilíneo cuyo destino es una transformación.
procedure MovRelCoor(destination:TTransformacion);	Movimiento coordinado relativo a la herramienta.
procedure MovRelLin(destination:TTransformacion);	Movimiento rectilíneo relativo a la herramienta.
procedure Drive(numero:integer;valor:single);	Mueve la articulación dada por el primer parámetro un ángulo/distancia dada por el segundo parámetro.
procedure Home;	Vuelve a la posición de reposo del robot.

Las siguientes instrucciones son relativas al uso de la herramienta asociada al robot. Si el robot no tuviera ninguna herramienta asociada y se utilizara alguna de las siguientes expresiones, dichas expresiones no tendrían ningún efecto.

procedure Tool(T:TTransformacion);	Cambia la referencia asociada a la herramienta, y por tanto, la referencia asociada al extremo del robot.
procedure Open;	Abre la herramienta hasta la apertura máxima.
procedure close(amount:single)	Cierra la herramienta hasta la apertura especificada por el parámetro (en mm).

Las siguientes instrucciones son relativas al uso de las entradas y salidas binarias asociadas al robot.

procedure wait(i:integer);	Detiene la ejecución del programa hasta que se active la entrada referenciada.
procedure signal(i:integer);	Activa o desactiva la salida referenciada dependiendo de su signo. Si el signo es positivo se activará la salida, mientras que si es negativo se desactivará.
function sig(i:integer):Boolean;	Lee el estado de la entrada tomada como parámetro.

7 RESULTADOS FINALES

Este apartado se refiere al conjunto de ensayos, búsquedas, errores, decisiones y avances que han permitido conseguir los objetivos propuestos para el Proyecto.

Está dividido en tres subapartados. En el primero de ellos se detallará la secuencia cronológica de acciones realizadas para alcanzar el objetivo final del Proyecto. En el segundo, se especificarán los principales problemas surgidos en la elaboración del proyecto. Finalmente se realizará un comentario general a modo de conclusión final acerca del proyecto.

7.1 SECUENCIA CRONOLÓGICA DE ACTUACIONES

Se describe a continuación la secuencia de pasos que han permitido conseguir el objetivo del Proyecto; se señalan además los principales problemas surgidos, cuyas soluciones respectivas se tratarán en el apartado siguiente.

Tarea 1: **movimiento de 2 robots añadidos a un mismo escenario**

1. Creación de vector de robots
2. Modificación de *ficheros.pas* para cargar robots al vector
3. Modificación de las instrucciones de *programa.pas*
4. Modificación del evento del *cadencer* para realizar movimientos de todos los robots dentro del vector.

Tarea 2: Creación y gestión de una ventana de programa para cada robot

5. Corrección de cambios en *programa.pas* para que la instrucción sólo mueva un robot.
6. Creación de vector de ventanas de programación.
7. Confección de un atributo *Script* en la clase *robot* y un atributo *robot* en la clase *Ventana_programacion*.
8. Instanciación de una ventana de programación por robot
9. Modificación de *ficheros.pas* para cargar y guardar los *scripts* de cada robot. Esto incluye cambios en la forma de guardar el *.esc,* distinguiendo y permitiendo la carga de versiones anteriores o actuales del *.esc*
10. Modificación de los *procedures* relativos a *eliminar robot* y *eliminar escenario* para cambiar también los vectores de robots y de programación

Tarea 3: Concurrencia de robots

- **Problema I:** No es posible ejecutar dos *scripts* a la vez.

- **Problema II:** Los *scripts* no se ejecutan sobre el robot correspondiente, se ejecutan sobre el último robot activo. En este paso se eliminan algunas dependencias carentes de sentido como que un programa contuviera un robot y que un robot contuviera a su vez un programa. El problema aparece debido a que al pasar las instrucciones por un *debugger* que lo ejecuta no utiliza la instancia que se ha creado para ello, por tanto la variable *robot* esta vacía.

11. Creación de un programa anexo (*cronometros*) para intentar ejecutar dos *scripts* a la vez. Pero cuando un *script* se está ejecutando toma posesión de la aplicación. Aparentemente no se consigue concurrencia al *instanciar* las ventanas de programación.

12. Para solucionar el problema se plantea la idea de migrar a otras librerías diferentes a *pascalscript.* Se estudia mejor el funcionamiento de *pascalscript*. Se empieza a hablar del *multitask* y *multitarea*.
13. Constatación de que *Delphi,* en principio, no es multitarea. Estudio de la clase *TThread* como posible solución al problema, ya que permite crear varios hilos de ejecución dentro de una aplicación en *Delphi*. Se descubre un componente que permite diferentes hilos de ejecución de manera sencilla pero se descarta puesto que sólo sirve para ejecutar la misma tarea repetidas veces a la vez y se necesita que cada *script* se ejecute independientemente.
14. Estudio de diferentes ejemplos y manuales del uso de la clase *TThread* y sincronización entre diferentes hilos de ejecución.
15. Modificación del programa *cronometros* de forma que se consigan ejecutar varios *scripts* simultáneamente.
16. Creación de lanzadores de instrucciones dentro de los hilos por dos motivos:
    - La sincronización de algunas instrucciones problemáticas (es necesario meter *Synchronize* dentro del hilo).
    - Para poder solucionar el problema I y poder llamar a la instancia de *Tprograma* creada que contendrá a su robot correspondiente, permitiendo así que cada robot ejecute su *script* de programación.
17. Integración del programa *cronometros* a *RobotScene* y sincronización de gráficos con la VCL de Windows para evitar fallos al mostrar mensajes.

- **Problema III:** Al integrar el programa *cronometros* a *RobotScene* hay que modificar el evento *OnIdle* del *debugger* para no llamar al *cadencer* pues, de lo contrario, perdemos la concurrencia. Al quitarlo aparecen errores con el uso de la garra que se solucionan introduciendo un *sleep* en el evento *OnIdle*.

  Se descubre que ocurren también errores al dejar las cajas pues, en ocasiones, no se depositan en el lugar adecuado. A veces el robot arrastra una caja que no debiera. Se detecta y corrige un fallo a la hora de comprobar colisiones, se comprueban siempre para todos los robots cuando deben comprobarse sólo para el robot que esta cerrando su garra.

  Se intenta solucionar en vano probando diferentes acciones: Sincronizando las instrucciones del hilo, creando un evento que dispare el *cadencer* cuando el hilo ejecuta una instrucción, pausando los hilos mientras el *cadencer* esta funcionando. Metiendo el *checkcollision* dentro de *herramienta.mover,* esperando en los hilos hasta que se producen dos disparos del *cadencer*, probando diferentes tiempos de espera, ajustando diferentes prioridades a los hilos.

18. Análisis de las dependencias entre las diferentes clases que se usan en Robotscene, limpiando el código y evitando variables repetidas.
19. Modificación de *robot.pas* para sincronizar mensajes de error.

20. Modificación del programa para que cada ventana de programación muestre en su *caption* el nombre de su correspondiente robot.
21. El problema era debido a que se modificaba el estado del robot antes de que se calcularan sus posiciones finales. Al modificarse el estado del robot, el *cadencer* empieza a dibujar las nuevas posiciones del robot antes de tiempo, lo que producía una finalización apresurada del movimiento. Solución del fallo de la herramienta: al cargar una herramienta fallaba si previamente ya había sido cargada una, no se eliminaban correctamente los ejes de la antigua herramienta.
22. Modificación de la forma de visualizar los ejes de los objetos y robots, al pulsar en el botón de los ejes aparecen todos los ejes importantes del escenario.

Tarea 4: Comunicación entre robots.

23. Creación de un nuevo formulario para la parte de comunicación entre robots. La idea es un entorno gráfico que permita simular un cableado entre robots. Para ello se utiliza *GLScene* y diferentes componentes (botones, planos y lineas).
24. Ajuste de la colocación de botones y del tamaño de algunos componentes gráficos a las dimensiones de pantalla del ordenador en uso.
25. Creación, mediante eventos de *drag and drop,* de cables arrastrando desde un *pad* de salida a un *pad* de entrada.
26. Se permite la eliminación de cables presionando click derecho del ratón sobre un cable.
27. Tras la finalización de la parte gráfica, creación de *matrices*, para guardar las conexiones, que serán utilizadas posteriormente.
28. Integración del nuevo formulario en *RobotScene*.
29. La imagen de estilo de formulario se guarda como imagen persistente dentro de la aplicación, para evitar un problema con las rutas de archivo porque fallaba al buscar la ruta.
30. Modificación de instrucciones del *script* para que al activar/desactivar la salida de un robot, se activen/desactiven las entradas cableadas, permitiendo la comunicación entre los robots mediante las instrucciones *signal()* y *wait()*.
31. Creación de instrucciones para permitir la carga y almacenado del cableado en un archivo *.cbl*. A su vez se crean botones para manejar la carga y la grabación del archivo *.cbl*.
32. Modificación en *ficheros.pas* la forma de cargar y guardar escenario para guardar en el archivo *.esc* la ruta del archivo *.cbl* y permitir así la carga del cableado al cargar el escenario
33. Modificación de las matrices cuando un robot se destruye y eliminación de cables con principio o final en ese robot.

34. Modificación de la ventana de cableado para que se visualicen únicamente las entradas y salidas de los robots existentes. El tamaño de la ventana se ajusta en correspondencia al número de robots.
35. Integración de la antigua ventana E/S para la monitorización de entradas y salidas. El tamaño de la ventana se ajusta automáticamente en correspondencia con el número de robots.
36. Adición de algunas instrucciones en el lenguaje robótico que no estaban correctamente asociadas.
37. Modificación y arreglo de *bugs* en el menú principal. Adicionalmente, se añaden opciones para poder acceder a la ventana de cableado desde el menú
38. Modificación de la barra de estado de la ventana de escenarios para mostrar las posiciones y orientaciones de las herramientas de los tres robots.
39. Realización de la Memoria.
40. Realización del manual de instrucciones.
41. Creación de un instalador de la aplicación.
42. Revisión general.

7.2 PROBLEMAS Y SOLUCIONES

En este apartado se analizan los principales problemas surgidos en el desarrollo de la versión multi-robot de **RobotScene**. Se ha querido detallar de forma concreta estos problemas porque han supuesto un incremento considerable en el tiempo previsto para el desarrollo del Proyecto, bien porque la solución no ha sido sencilla de encontrar, o bien porque, en algunos problemas, el método resolutivo no es sencillo de implementar.

El tratamiento y la resolución de los problemas surgidos en el desarrollo del Proyecto, recogidos en la secuencia cronológica de actuaciones, se explica a continuación.

Problema I: Imposibilidad de ejecutar dos scripts de programación de forma simultánea.

Ha constituido uno de los problemas más difíciles de solucionar en la versión multi-robot de **RobotScene**. Durante el desarrollo del Proyecto se supuso que, simplemente instanciando la ventana de programación, se podrían ejecutar dos scripts a la vez. Esta suposición se basaba en que al instanciar la ventana de programación realmente se estaba creando una copia de dicha ventana, y por lo tanto, se crea una copia del componente *TPScriptDebugger*.

Como esta copia del *debugger* funciona mediante eventos, se supuso que Delphi sería capaz de manejar dichos eventos y repartir equitativamente el tiempo entre los eventos. Sin embargo, este planteamiento estaba equivocado porque uno de los eventos de dicho componente se ejecuta de manera repetitiva hasta que se termina de ejecutar la última instrucción del programa robótico.

Para resolver este problema se buscaron algunas soluciones poco funcionales. Algunas de ellas se analizan a continuación:

Primero, se intentaron buscar soluciones sencillas, como por ejemplo pausar y resumir los scripts conforme se ejecutara una instrucción. Sin embargo, uno de los eventos del *debugger* se apoderaba de todo el flujo de ejecución del programa.

En este punto se decidió crear un programa independiente del simulador robótico, con la idea de ejecutar dos scripts de forma concurrente, para realizar pruebas de programación y asegurarnos de que el problema no era provocado por parte del código de **RobotScene**.

Tras comprobar que la ejecución de dos scripts simultáneamente iba a requerir más de un flujo de programación, se buscaron nuevas soluciones. Una de estas soluciones fue la búsqueda de unas librerías semejantes a Pascal Script que estuvieran pensadas de forma explícita para el uso de varios scripts a la vez. Esta solución quedó rechazada porque no existen unas librerías con estas características y, además, migrar de librerías supondría rehacer gran parte de la aplicación sin que existiera seguridad plena de haber resuelto el problema.

Otra de estas soluciones consistió en la búsqueda de un componente que permitiese crear hilos de manera sencilla. Sin embargo, estos componentes adolecen de limitaciones en algunos aspectos y no eran útiles para nuestro caso concreto.

Para resolver este problema se decide crear varios hilos de ejecución mediante la clase *TThread* de Delphi. Se estudian ejemplos obtenidos de los creadores de PascalScript (RemObjects) donde se utiliza el componente *TPScriptDebugger* en diferentes hilos.

Problema II: Los script se ejecutan sobre el último robot seleccionado en vez de sobre su robot correspondiente.

Este problema aparece de forma simultánea al primer problema. En ocasiones se llega a pensar, de forma equivocada, que este problema es la causa de no poder ejecutar dos scripts de forma concurrente. Sin embargo, tras la implementación de diferentes hilos de ejecución resulta sencillo resolverlo.

La causa de este problema reside en que al ejecutar instrucciones robóticas se llama a una instancia de *Tprograma* en vez de llamar a la variable de tipo *Tprograma* creada para llevar a cabo tal propósito. Muchas de las instrucciones contenidas en la clase *Tprograma* hacen uso de la variable *robot* de tipo *TRobot*. Si se utiliza una instancia nueva de *Tprograma* su variable robot esta vacía y el script no puede ejecutarse sobre el robot que queremos.

Para solucionar este problema se implementan dentro de los hilos lanzadores que realicen la llamada a las variables *programa* en vez de a las instancias de *Tprograma*. Esta implementación pudo observarse en el apartado relativo al código fuente de la aplicación.

Problema III: Los robots realizan movimientos extraños a la hora de depositar objetos agarrados mediante la herramienta.

Cuando se consiguió la ejecución de cada script sobre su correspondiente robot, se descubrió que algo fallaba en los movimientos de los robots. Lo primero que se observa son fallos con el manejo de la herramienta pues, en algunos ocasiones, el robot no responde a las peticiones de apertura y cierre de la garra. Por otro lado, el robot a veces arrastraba objetos que no debía. Por último, el robot a veces finalizaba antes de tiempo algunos movimientos soltando los objetos en lugares equivocados.

La dificultad a la hora de solucionar este problema consistió en encontrar el error de programación sin poder seguir un flujo de programa. Esto era debido a que en ese momento el programa maneja varios hilos de ejecución y por tanto, los errores se vuelven difíciles de detectar, ya que no ocurren siempre sino que lo hacen de manera aleatoria.

Para solucionar el primero de los fallos en el movimiento, se introduce una instrucción *sleep* en el evento del *debugger*. Con este retardo se consigue, por un lado, que el robot responda a las peticiones de apertura y cierre y, por otro lado,

suspender por un tiempo la ejecución del hilo secundario con la consiguiente rebaja de carga de trabajo en el procesador.

El segundo error se soluciona modificando las instrucciones relativas a la comprobación de colisiones con la garra. Las colisiones se comprobaban cuando un robot agarraba un objeto, pero para todos los robots del escenario. Se corrigió este error para que la detección de colisiones se realizara únicamente sobre el robot que estaba cerrando su herramienta.

El tercero de los problemas fue el más difícil de detectar; sin embargo la solución era tan fácil como cambiar de lugar una instrucción. Como se ha comentado anteriormente los movimientos del robot se ejecutan bajo un sistema de eventos discretos. La instrucción que cambia el estado del robot en movimientos articulares y rectilíneos se realizaba antes de la finalización del movimiento. En la versión mono-robot este problema no afectaba porque el programa seguía un flujo determinado. Sin embargo se hizo presente cuando se ejecutaba a la vez el procedimiento con dicha instrucción y el evento del *cadencer*. Como se ha comentado, la solución consistía en cambiar de lugar la instrucción que modifica el estado del robot.

7.3 COMENTARIOS FINALES

En este apartado se quiere realizar una serie de comentarios finales a modo de conclusión. Llegados a este punto ya se conocen las características y el desarrollo de la nueva versión de **RobotScene**. Con esta versión se pretendía dotar al simulador con la capacidad de crear y explotar escenarios robóticos que incluyesen varios robots. Por tanto, se puede afirmar que tanto el objetivo principal como el resto de requisitos han sido cumplidos.

Por otro lado, se quiere remarcar en estas últimas líneas la dificultad del Proyecto por varias razones. Trabajar con el código fuente de una aplicación escrita por otras personas añade dificultad al proyecto puesto que es necesario entender las instrucciones escritas previamente y, en cierto modo, penetrar en la forma de pensar de quien ha estructurado y escrito el código. Aunque en algunas ocasiones contar con un código de partida ahorra tiempo al tener partes de código ya escritas, en otras ocasiones se pierde un tiempo muy valioso al tener que detectar errores en código que no ha elaborado la persona que lo modifica.

Además, el trabajo con diferentes hilos de ejecución es una tarea compleja. Más aún, cuando no se ha programado antes con lenguajes de programación orientados a objetos, sino únicamente con lenguajes estructurados. La programación concurrente o multi-hilo entraña una serie de dificultades. Por ejemplo, se deben sincronizar con la tarea principal las instrucciones que entrañen algún riesgo de entrar es secciones críticas o que utilicen variables fuera del hilo de ejecución secundario.

Pese a todo lo anterior, la mayor dificultad en la programación con varios hilos es que los errores de programación son muy difíciles de detectar. Esto se debe a que el programa no sigue un único flujo de ejecución y ante diferentes ejecuciones de un programa, éste último no se comporta de la misma manera. Los errores salen a la luz de forma aleatoria según se producen unas determinadas y específicas circunstancias.

8 REFERENCIAS BIBLIOGRÁFICAS E INFORMÁTICAS

La preparación, elaboración y desarrollo del Proyecto ha necesitado la consulta de los textos, libros, manuales y sitios web que se relacionan a continuación. Por otra parte, ha sido preciso utilizar un conjunto de herramientas informáticas, modelos y programas que aparecen también reseñadas.

8.1 BIBLIOGRAFÍA / LINKOGRAFÍA

Textos consultados

Memoria PFC. Software de creación y explotación de escenarios robóticos. *Javier Fermín Hernández.*

Manual de instrucciones RobotScene. Software de creación y explotación de escenarios robóticos. *Javier Fermín Hernández*.

Apuntes de la asignatura Robótica Industrial. 3º Ingeniería Técnica Industrial. Especialidad Electrónica Industrial. *Escuela Universitaria de Ingeniería Técnica Industrial (Departamento Ingeniería de Sistemas y Automática).*

Proyectos sobre Soportes Digitales con Capacidades Interactivas y Multimedia. *Torres, Manuel; Torres Miguel A*. Escuela Universitaria de Ingeniería Técnica Industrial (Departamento Ingeniería de Diseño y Fabricación). Curso 2004-2005.

Guía de iniciación al GLScene. *David Martín de Vidales*.

Libros consultados

La Cara Oculta de Delphi 6. *Ian Marteens*. Intuitive Sight S.L. 2002. ISBN 9788460738732.

La Biblia de Delphi 7. *Marco Cantú*. Anaya Multimedia. 2003. ISBN 9788441515703.

Fundamentos de Robótica. *Antonio Barrientos, Luís Felipe Peñín, Carlos Balaguer, Rafael Aracil*. Universidad Politécnica de Madrid. McGRAW-HILL / INTERAMERICANA DE ESPAÑA S.A. 1997. ISBN 84-481-0815-9.

Direcciones web de interés

http://es.wikipedia.org/ Wikipedia, la enciclopedia libre.

http://en.wikipedia.org/ Wikipedia. Versión inglesa.

http://www.rae.es/rae.html Real Academia Española.

http://www.clubdelphi.com/ Punto de encuentro de los programadores en Delpi.

http://wiki.remobjects.com/wiki/Main_Page RemObjects Wiki.

http://www.robodosis.com/2012/03/historia-de-la-robotica.html Robodosis, blog dedicado a la robótica.

http://automata.cps.unizar.es/Historia/Webs/automatas_en_la_historia.htm Autómatas en la historia. Unizar.

http://www.madehow.com/Volume-2/Industrial-Robot.html Historia de Robótica Industrial.

http://www.newtonium.com/public_html/Products/RoboWorks/RoboWorks.htm Páginal web del simulador RoboWorks.

http://www.openrtp.jp/openhrp3/en/index.html Página web del proyecto hrp3.

http://www.gazebosim.org/ Página web del simulador Gazebo.

http://www.parallemic.org/RoKiSim.html Página web del Simulador RoKiSim.

http://simbad.sourceforge.net/ Página web del simulador Simbad.

http://www.robologix.com/ Página web del simulador Robologix.

http://www.cyberbotics.com/ Página web del simulador Webots.

http://www.workspace5.com/ Página web del simulador Workspace.

http://www.staubli.es Página web del fabricante de robots Staubli.

Direcciones web de interés sobre normativa del sector electrónico

http://www.aenor.es Aenor es la asociación española de normalización y certificación.

http://www.iso.ch/welcome.html Es la página principal de ISO. Organización Internacional de Estandarización.

http://www.din.de Página principal de DIN, en Alemania.

8.2 PROGRAMAS DE CÁLCULO Y SOFTWARE UTILIZADOS

Delphi 7. *Bordland Software Corporation*. 2002.

GLScene v1.0.0.0714. *Mike Lischke, Eric Grange*. 2006.

RemObjects PascalScript for Delphi 3.0.47.841. *RemObjects*. 2005.

SynEdit 2.0.6. *alkisg, harmeister*. 2007

9 GLOSARIO DE TÉRMINOS

Se definen a continuación algunos términos y expresiones técnicas utilizados en la redacción de la Memoria del Proyecto. De esta manera se facilita su comprensión.

9.1 DEFINICIONES

Blog	*Sitio web periódicamente actualizado que recopila cronológicamente textos o artículos de uno o varios autores que conservan siempre la libertad de dejar publicado lo que crean pertinente.*
Buffer	*Un* buffer *(o búfer) en informática es un espacio de memoria, en el que se almacenan datos para evitar que el programa o recurso que los requiere, ya sea hardware o software, se quede sin datos durante una transferencia.*
CD	*Disco óptico que se graba en forma digital, lo que permite acumular una gran cantidad de información.*
Clase	*En la programación orientada a objetos, una* clase *es una construcción que se utiliza como un modelo (o plantilla) para crear objetos de ese tipo. El modelo describe el estado y el comportamiento que todos los objetos de la clase comparten. Un objeto de una determinada clase se denomina una instancia de la clase. La clase que contiene (y se utilizó para crear) esa instancia se puede considerar como del tipo de ese objeto, por ejemplo, una instancia del objeto de la clase "Persona" sería del tipo "Persona".*
Código fuente	*El código fuente de un programa informático (o software) es un conjunto de líneas de texto que constituyen las instrucciones que debe seguir la computadora para ejecutar dicho programa. Por tanto, en el código fuente de un programa está descrito por completo su funcionamiento.*
Driver	*También denominado controlador de dispositivo. Es un programa informático que permite al sistema operativo interactuar con un periférico, haciendo una abstracción del hardware y proporcionando una interfaz -posiblemente estandarizada- para usarlo. Se puede esquematizar como un manual de instrucciones que le indica al sistema operativo, cómo debe controlar y comunicarse con un dispositivo en particular. Por tanto, es una pieza esencial, sin la cual no se podría usar el hardware.*

Electrónica	*Estudio y aplicación del comportamiento de los electrones en diversos medios, como el vacío, los gases y los semiconductores, sometidos a la acción de campos eléctricos y magnéticos.*
Función	*En computación, una* función *o subrutina (también llamada procedimiento, o rutina).Se presenta como un subalgoritmo que forma parte del algoritmo principal, el cual permite resolver una tarea específica.*
Hardware	*Corresponde a todas las partes tangibles de un sistema informático. Sus componentes son: eléctricos, electrónicos, electromecánicos y mecánicos. Son cables, gabinetes o cajas, periféricos de todo tipo y cualquier otro elemento físico involucrado; contrariamente, el soporte lógico es intangible y es llamado software.*
Hilo de ejecución	*En sistemas operativos, un* hilo de ejecución*, hebra o subproceso es la unidad de procesamiento más pequeña que puede ser planificada por el sistema operativo. La creación de un nuevo hilo es una característica que permite a una aplicación realizar varias tareas a la vez (concurrentemente). Los distintos hilos de ejecución comparten una serie de recursos tales como el espacio de memoria, los archivos abiertos, situación de autenticación, etc. Esta técnica permite simplificar el diseño de una aplicación que debe llevar a cabo distintas funciones simultáneamente.*
Ingeniería electrónica	*Conjunto de conocimientos técnicos, tanto teóricos como prácticos que tienen por objetivo la aplicación de la tecnología electrónica para la resolución de problemas prácticos.*
Instancia	*La palabra Instancia significa: Solicitud o Insistencia. Una* instancia *de un programa es una copia de una versión ejecutable del programa que ha sido escrito en la memoria del computador. Instancia puede referirse al objeto que almacena los datos de una clase.*
Lenguaje de programación	*Un lenguaje de programación es un idioma artificial diseñado para expresar procesos que pueden ser llevadas a cabo por máquinas como las computadoras. Pueden usarse para crear programas que controlen el comportamiento físico y lógico de una máquina, para expresar algoritmos con precisión, o como modo de comunicación humana.*

Librería	*En ciencias de la computación, una* librería *o biblioteca (del inglés library) es un conjunto de subprogramas utilizados para desarrollar software. Las bibliotecas contienen código y datos, que proporcionan servicios a programas independientes, es decir, pasan a formar parte de éstos. Esto permite que el código y los datos se compartan y puedan modificarse de forma modular.*
Manipulador	*Un manipulador es un mecanismo utilizado bajo control humano para manipular materiales sin establecer un contacto directo. Estos materiales son con frecuencia radiactivos, suponen un riesgo biológico o bien se encuentran situados en lugares inaccesibles.*
Motor gráfico	*Es un término que hace referencia a una serie de rutinas de programación que permiten el diseño, la creación y la representación de escenarios gráficos. La funcionalidad básica de un motor es proveer a la aplicación informática de un motor de renderizado para los gráficos 2D y 3D, motor físico o detector de colisiones, sonidos, animación, inteligencia artificial, redes, streaming, administración de memoria y un escenario gráfico.*
Página web	*Una página web es el nombre de un documento o información electrónica adaptada para la World Wide Web al que se puede acceder mediante un navegador. Esta información se encuentra generalmente en formato HTML o XHTML, y puede proporcionar navegación a otras páginas web mediante enlaces de hipertexto.*
Programa informático	*Un programa informático es un conjunto de instrucciones que una vez ejecutadas realizarán una o varias tareas en una computadora.*
Programación dirigida por eventos	*La programación dirigida por eventos es un paradigma de programación en el que, tanto la estructura como la ejecución de los programas, van determinados por los sucesos que ocurran en el sistema, definidos por el usuario o que ellos mismos provoquen.*
Programación concurrente	*La programación concurrente es la simultaneidad en la ejecución de múltiples tareas interactivas. Estas tareas pueden ser un conjunto de procesos o hilos de ejecución creados por un único programa.*
Programación estructurada	*La programación estructurada es una técnica para escribir programas. En ella se utilizan únicamente tres estructuras: secuencia, selección e iteración; siendo innecesario el uso de la instrucción o instrucciones de transferencia incondicional.*

Programación orientada a objetos	*La programación orientada a objetos o POO (OOP según sus siglas en inglés) es un paradigma de programación que usa los objetos en sus interacciones, para diseñar aplicaciones y programas informáticos. Está basado en varias técnicas, incluyendo herencia, cohesión, abstracción, polimorfismo, acoplamiento y encapsulamiento.*
Script	*En informática un guión, archivo de órdenes o archivo de procesamiento por lotes. Es un programa usualmente simple, que por lo regular se almacena en un archivo de texto plano. El uso habitual de los guiones es realizar diversas tareas como combinar componentes, interactuar con el sistema operativo o con el usuario.*
Sensor	*Dispositivo capaz de detectar magnitudes físicas o químicas, llamadas variables de instrumentación, y transformarlas en variables eléctricas.*
Sistema operativo	*Un sistema operativo (SO) es un programa o conjunto de programas que en un sistema informático gestiona los recursos de hardware y provee servicios a los programas de aplicación, ejecutándose en modo privilegiado respecto de los restantes.*
Software	*La palabra software se refiere al equipamiento lógico o soporte lógico de un computador digital, y comprende el conjunto de los componentes lógicos necesarios para hacer posible la realización de una tarea específica, en contraposición a los componentes físicos del sistema (hardware).*
Sprite	*Un sprite es un tipo de mapa de bits dibujados en la pantalla de ordenador por hardware gráfico especializado sin cálculos adicionales de la CPU. A menudo son pequeños y parcialmente transparentes, dejándoles así asumir otras formas a la del rectángulo.*
Teleoperación	*La teleoperación significa "hacer el trabajo a distancia", aunque "trabajo" puede significar casi cualquier cosa. Además, el termino "distancia" es vago: puede referirse a una distancia física, donde el operador esta separado del robot por una larga distancia, pero puede también referirse a un cambio de escala, donde por ejemplo en cirugía robótica un cirujano puede usar tecnología de micro-manipulación para dirigir cirugías a nivel microscópico.*

Textura	*Una textura es una imagen del tipo bitmap utilizada para cubrir la superficie de un objeto virtual, ya sea tridimensional o bidimensional, con un programa de gráficos especial*
Unidad central de procesamiento (CPU o procesador)	*es el componente en una computadora digital que interpreta las instrucciones y procesa los datos contenidos en los programas de la computadora.*
Variable	*En programación, las variables son espacios reservados en la memoria que, como su nombre indica, pueden cambiar de contenido a lo largo de la ejecución de un programa. Una variable corresponde a un área reservada en la memoria principal del ordenador pudiendo ser de longitud fija o variable.*
Wiki	*Un wiki o una wiki es un sitio web cuyas páginas pueden ser editadas por múltiples voluntarios a través del navegador web. Los usuarios pueden crear, modificar o borrar un mismo texto que comparten.*

9.2 ABREVIATURAS

API	Application Programming Interface (interfaz de programación de aplicaciones).
CD	Compact Disc (disco compacto).
CPU	Central Processing Unit (unidad central de procesamiento).
GPU	Graphic Process Unit (unidad de procesamiento de gráficos).
DVD	Digital Versatile Disc (disco versátil digital).
E/S	Entrada / Salida.
IEC	International Electrotechnical Commission (Comisión Electrónica Internacional).
ISO	International Organization for Standardization (Organización Internacional para la Estandarización).
VCL	Visual Component Library (biblioteca de componentes visuales).